Tr-êng §¹i häc b¸ch khoa Hµ Néi
Khoa c«ng nghÖ th«ng tin
Bé m«n kü thuËt m¸y tÝnh

---

b¸o c¸o ®å ¸n m«n häc
ThiÕt kÕ m¹ch nhê m¸y tÝnh

§Ò tµi:

ThiÕt kÕ m¹ch b»ng VHDL

Giáo viên hướng dẫn: ***th.s.*** nguyÔn phó b×nh

Nhóm sinh viên thực hiện: Lª tuÊn anh
Nghiªm kim ph-¬ng
NguyÔn quèc viÖt
NguyÔn ngäc linh

Lớp: ktmt - K46

Hà Nội, 10/2005

Mục lục

Trang

Danh mục hình:

Trang

Danh mục bảng:

Trang

Chương 1: Giới thiệu

1.1. Giới thiệu về VHDL

VHDL là ngôn ngữ mô tả phần cứng cho các mạch tích hợp tốc độ rất cao, là một loại ngôn ngữ mô tả phần cứng được phát triển dùng cho trương trình VHSIC(Very High Speed Itergrated Circuit) của bộ quốc phòng Mỹ. Mục tiêu của việc phát triển VHDL là có được một ngôn ngữ mô phỏng phần cứng tiêu chuẩn và thống nhất cho phép thử nghiệm các hệ thống số nhanh hơn cũng như cho phép dễ dàng đưa các hệ thống đó vào ứng dụng trong thực tế. Ngôn ngữ VHDL được ba công ty Intermetics, IBM và Texas Instruments bắt đầu nghiên cứu phát triển vào tháng 7 năm 1983. Phiên bản đầu tiên được công bố vào tháng 8-1985. Sau đó VHDL được đề xuất để tổ chức IEEE xem xét thành một tiêu chuẩn chung. Năm 1987 đã đưa ra tiêu chuẩn về VHDL(tiêu chuẩn IEEE-1076-1987).

VHDL được phát triển để giải quyết các khó khăn trong việc phát triển, thay đổi và lập tài liệu cho các hệ thống số. VHDL là một ngôn ngữ độc lập không gắn với bất kỳ một phương pháp thiết kế, một bộ mô tả hay công nghệ phần cứng nào. Người thiết kế có thể tự do lựa chọn công nghệ, phương pháp thiết kế trong khi chỉ sử dụng một ngôn ngữ duy nhất. Và khi đem so sánh với các ngôn ngữ mô phỏng phần cứng khác ta thấy VHDL có một số ưu điểm hơn hẳn là:

- *Thứ nhất là tính công cộng:*

VHDL được phát triển dưới sự bảo trợ của chính phủ Mỹ và hiện nay là một tiêu chuẩn của IEEE. VHDL được sự hỗ trợ của nhiều nhà sản xuất thiết bị cũng như nhiều nhà cung cấp công cụ thiết kế mô phỏng hệ thống.

- *Thứ hai là khả năng được hỗ trợ bởi nhiều công nghệ và nhiều phương pháp thiết kế*:

VHDL cho phép thiết kế bằng nhiều phương pháp ví dụ phương pháp thiết kế từ trên xuống, hay từ dưới lên dựa vào các thư viện sẵn có. VHDL cũng hỗ trợ cho nhiều loại công cụ xây dựng mạch như sử dụng công nghệ đồng bộ hay không đồng bộ, sử dụng ma trận lập trình được hay sử dụng mảng ngẫu nhiên.

- *Thứ ba là tính độc lập với công nghệ:*

VHDL hoàn toàn độc lập với công nghệ chế tạo phần cứng. Một mô tả hệ thống dùng VHDL thiết kế ở mức cổng có thể được chuyển thành các bản tổng hợp mạch khác nhau tuỳ thuộc công nghệ chế tạo phần cứng mới ra đời nó có thể được áp dụng ngay cho các hệ thống đã thiết kế.

- *Thứ tư là khả năng mô tả mở rộng:*

VHDL cho phép mô tả hoạt động của phần cứng từ mức hệ thống số cho đến mức cổng. VHDL có khả năng mô tả hoạt động của hệ thống trên nhiều

mức nhưng chỉ sử dụng một cú pháp chặt chẽ thống nhất cho mọi mức. Như thế ta có thể mô phỏng một bản thiết kế bao gồm cả các hệ con được mô tả chi tiết.

- *Thứ năm là khả năng trao đổi kết quả:*

Vì VHDL là một tiêu chuẩn được chấp nhận, nên một mô hình VHDL có thể chạy trên mọi bộ mô tả đáp ứng được tiêu chuẩn VHDL. Các kết quả mô tả hệ thống có thể được trao đổi giữa các nhà thiết kế sử dụng công cụ thiết kế khác nhau nhưng cùng tuân theo tiêu chuẩn VHDL. Cũng như một nhóm thiết kế có thể trao đổi mô tả mức cao của các hệ thống con trong một hệ thống lớn (trong đó các hệ con đó được thiết kế độc lập).

- *Thứ sáu là khả năng hỗ trợ thiết kế mức lớn và khả năng sử dụng lại các thiết kế:*

VHDL được phát triển như một ngôn ngữ lập trình bậc cao, vì vậy nó có thể được sử dụng để thiết kế một hệ thống lớn với sự tham gia của một nhóm nhiều người. Bên trong ngôn ngữ VHDL có nhiều tính năng hỗ trợ việc quản lý, thử nghiệm và chia sẻ thiết kế. Và nó cũng cho phép dùng lại các phần đã có sẵn.

1.2. Giới thiệu công nghệ (và ứng dụng) thiết kế mạch bằng VHDL.

1.2.1 Ứng dụng của công nghệ thiết kế mạch bằng VHDL

Hiện nay 2 ứng dụng chính và trực tiếp của VHDL là các ứng dụng trong các thiết bị logic có thể lập trình được (Programmable Logic Devices – PLD) (bao gồm các thiết bị logic phức tạp có thể lập trình được và các FPGA - Field Programmable Gate Arrays) và ứng dụng trong ASICs(Application Specific Integrated Circuits).

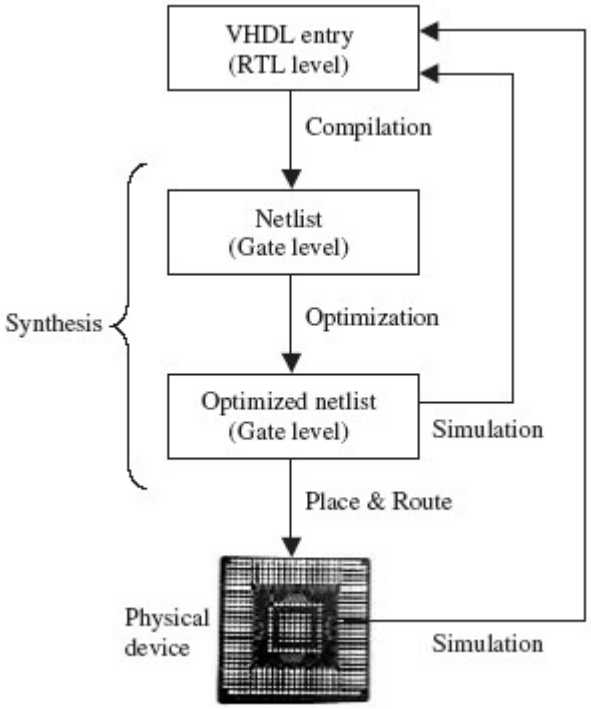
Khi chúng ta lập trình cho các thiết bị thì chúng ta chỉ cần viết mã VHDL một lần, sau đó ta có thể áp dụng cho các thiết bị khác nhau (như Altera, Xilinx, Atmel,…) hoặc có thể để chế tạo một con chip ASIC. Hiện nay, có nhiều thương mại phức tạp (như các vi điều khiển) được thiết kế theo dựa trên ngôn ngữ VHDL.

1.2.2 Quy trình thiết kế mạch bằng VHDL.

Như đề cập ở trên, một trong số lớn các ứng dụng của VHDL là chế tạo các mạch hoặc hệ thống trong thiết bị có thể lập trình được (PLD hoặc FPGA) hoặc trong ASIC. Việc chế tao ra vi mạch sẽ được chia thành 3 giai đoạn như sau:

- *Giai đoạn 1*:

Chúng ta bắt đầu thiết kế bằng viết mã VHDL. Mã VHDL này sẽ được lưu vào file có đuôi là .vhd và có tên cùng với tên thực thể. Mã VHDL sẽ được mô tả ở tầng chuyển đổi thanh ghi.



Hình 1.1. Tóm tắt quy trình thiết kế VHDL

- *Giai đoạn 2*: Giai đoạn chế tạo:

Bước đầu tiên trong quá trình chế tạo là biên dich. Quá trình biên dịch sẽ chuyển mã VHDL vào một netlist ở tầng cổng.

Bước thứ 2 của quá trình chế tạo là tối ưu. Quá trình tối ưu được thực hiện trên netlist ở tầng cổng về tốc độ và phạm vi.

Trong giai đoạn này, thiết kế có thể được mô phỏng để kiểm tra phát hiện những lỗi xảy ra trong quá trình chế tạo.

- *Giai đoạn 3*:

Là giai đoạn ghép nối đóng gói phần mềm. Ở giai đoạn này sẽ tạo ra sự sắp xếp vật lý cho chip PLD/FPGA hoặc tạo ra mặt nạ cho ASIC.

1.2.3. Công cụ EDA.

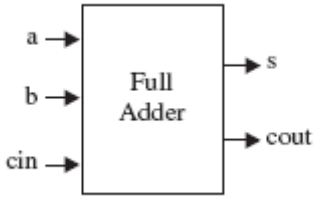
Các công cụ phục vụ cho quá trình thiết kế vi mạch sẽ là:

- Công cụ Active – HDL: Tạo mã VHDL và mô phỏng

- Công cụ EDA (Electronic Design Automation): là công cụ tự động thiết kế mạch điện tử. Công cụ này được dùng để phục vụ cho việc chế tạo, thực thi và mô phỏng mạch sử dụng VHDL.
- Công cụ cho đóng gói: Các công cụ này sẽ cho phép tổng hợp mã VHDL vào các chip CPLD/FPGA của Altera hoặc hệ ISE của Xilinx, for Xilinx's CPLD/FPGA chips).

1.2.4. Chuyển mã VHDL vào mạch.

Một bộ cộng đầy đủ được mô tả trong hình dưới đây:



Hình 1.2.a. Sơ đồ tổng quát về bộ cộng đầy đủ

Trong đó, a , b là các bit vào cho bộ cộng, cin là bit nhớ. Đầu ra s là bit tổng, cout là bit nhớ ra. Hoạt động của mạch được chỉ ra dưới dạng bảng chân lý:

a b cin	s cout
0 0 0	0 0
0 1 0	1 0
1 0 0	1 0
1 1 0	0 1
0 0 1	1 0
0 1 1	0 1
1 0 1	0 1
1 1 1	1 1

Hình 1.2.b. Bảng chân lý của bộ cộng đầy đủ

Bit s và cout được tính như sau:

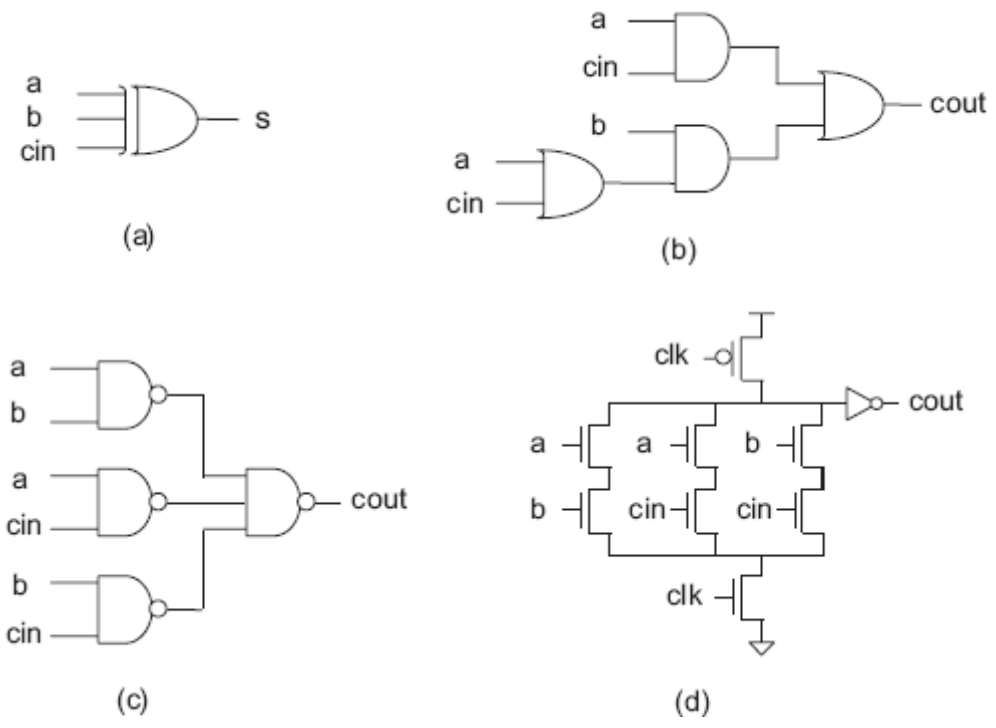
$s = a \oplus b \oplus cin$ và $cout = a.b + a.cin + b.cin$

Từ công thức tính s và cout ta viết đoạn mã VHDL như dưới đây:

```
5   library IEEE;
6   use IEEE.STD_LOGIC_1164.all;
7   entity FULL_ADDER_1 is
8       port(
9            a : in BIT;
10           b : in BIT;
11           cin : in BIT;
12           s : out BIT;
13           cout : out BIT
14           );
15  end FULL_ADDER_1;
16  --}} End of automatically maintained section
17  architecture ARC_FULL_ADDER1 of FULL_ADDER_1 is
18  begin
19      s<= a xor b xor cin ;
20      cout <= (a and b)or(a and cin)or (b and cin);
21  end ARC_FULL_ADDER1;
```

Hình 1.3. Mã thiết kế bộ cộng

Từ mã VHDL này, mạch vật lý được tạo ra. Tuy nhiên có nhiều cách để thực hiện phương trình được miêu tả trong ARCHITECTURE OF, vì vậy mạch thực tế sẽ phụ thuộc vào bộ biên dịch/bộ tối ưu đang được sử dụng và đặc biệt phụ thuộc mục đích công nghệ. Hình vẽ sau đây thể hiện một số dạng kiến trúc của mạch cộng:

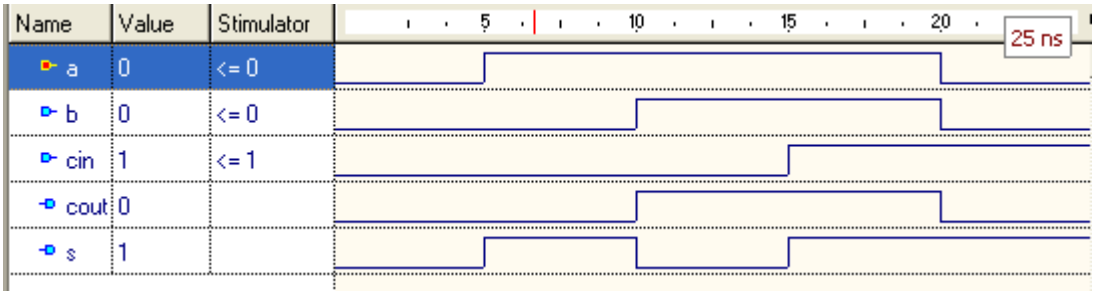


Hình 1.4.a. Các ví dụ về sơ đồ mạch có thể có ứng với mã như hình 1.3

Trong trường hợp này, nếu mục đích công nghệ của chúng ta là thiết bị lgic có thê lập trình được (PLD, FPGA), thì 2 kết quả cho cout thoả mãn là ở hình (b) và hình (c) ($cout = a.b + a.cin + b.cin$). Còn nếu mục đích công nghệ là ASIC, thì chúng ta có thể sử dụng hình (d). Hình D sử dụng công nghệ CMOS với các tầng transistor và các mặt nạ phủ.

Bất cứ một cái mạch nào được tao ra từ mã, thì những thao tác của nó sẽ luôn luôn được kiểm tra ở mức thiết kế, như ta đã chỉ ra ở hình 1. Tất nhiên, chúng ta cũng có thể kiểm tra nó ở tầng vật lý, nhưng sau đó những thay đổi là rất tai hại.

Hình dưới đây là mô phỏng kết quả của đoạn chương trình đã viết ở trên cho mạch bộ cộng đầy đủ ở hình 1.3.



Hình 1.4.b: Kết quả mô phỏng bộ cộng được thiết kế theo hình 1.3

Chương 2. Cấu trúc mã

Trong chương này, chúng ta mô tả các phần cơ bản có chứa cả các đoạn Code nhỏ của VHDL: các khai báo LIBRARY, ENTITY và ARCHITECTURE.

2.1. Các đơn vị VHDL cơ bản.

Một đoạn Code chuẩn của VHDL gồm tối thiểu 3 mục sau:

- Khai báo LIBRARY: chứa một danh sách của tất cả các thư viện được sử dụng trong thiết kế. Ví dụ: ieee, std, work, …
- ENTITY: Mô tả các chân vào ra (I/O pins) của mạch
- ARCHITECTURE: chứa mã VHDL, mô tả mạch sẽ hoạt động như thế nào.

Một LIBRARY là một tập các đoạn Code thường được sử dụng. Việc có một thư viện như vậy cho phép chúng được tái sử dụng và được chia sẻ cho các ứng dụng khác. Mã thường được viết theo các định dạng của FUNCTIONS, PROCEDURES, hoặc COMPONENTS, được thay thế bên trong PACKAGES và sau đó được dịch thành thư viện đích.

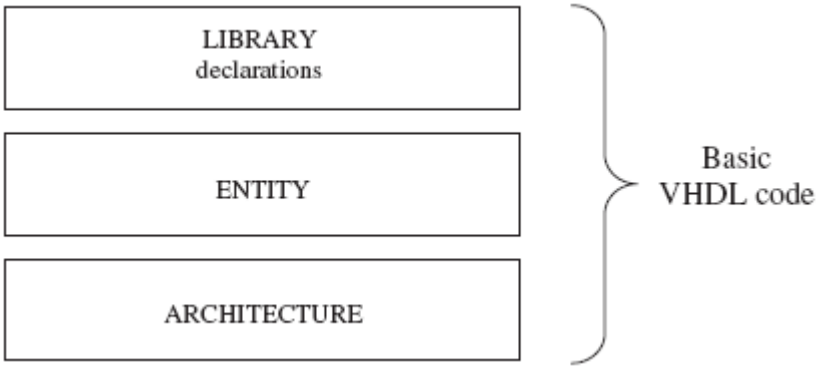
2.2. Khai báo Library.

- Để khai báo Library, chúng ta cần hai dòng mã sau, dòng thứ nhất chứa tên thư viện, dòng tiếp theo chứa một mệnh đề cần sử dụng:

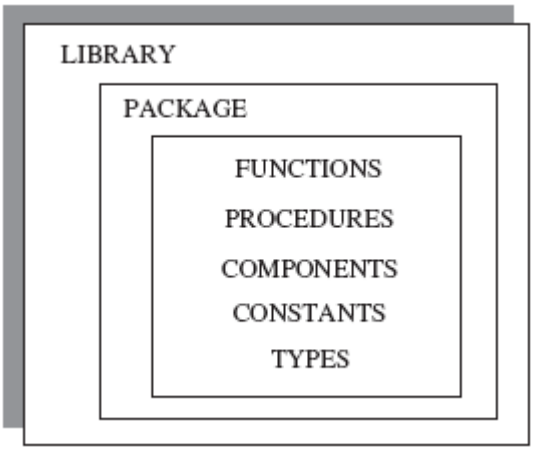
```
LIBRARY library_name;
USE library_name.package_name.package_parts;
```

Thông thường có 3 gói, từ 3 thư viện khác nhau thường được sử dụng trong thiết kế:

- ieee.std_logic_1164 (from the ieee library),
- standard (from the std library), and
- work (work library).



Hình 2.1: Các thành phần cơ bản của một đoạn mã VHDL



Hình 2.2: Các phần cơ bản của một Library

Các khai báo như sau:

```
LIBRARY ieee;            -- Dấu chấm phẩy (;) chỉ thị
USE ieee.std_logic_1164.all;-- kt của một câu lệnh

LIBRARY std;      -- hoặc một khai báo.một dấu 2 gạch
USE std.standard.all;   -- (--)để bắt đầu 1 chú thích.

LIBRARY work;
USE work.all;
```

Các thư viện *std* và *work* thường là mặc định, vì thế không cần khai báo chúng, chỉ có thư viện *ieee* là cần phải được viết rõ ra.

Mục đích của 3 gói/thư viện được kể ở trên là như sau: gói *std_logic_1164* của thư viện *ieee* cho biết một hệ logic đa mức; *std* là một thư viện tài nguyên (kiểu dữ kiệu, i/o text..) cho môi trường thiết kế VHDL và thư viện work được sủ dụng khi chúng ta lưu thiết kế (file .vhd, các file được tạop bởi chương trình dịch và chương trình mô phỏng…).

Thực ra, thư viện ieee chứa nhiều gói như sau:

- *std_logic_1164*: định rõ STD_LOGIC (8 mức) và STD_ULOGIC (9 mức) là các hệ logic đa mức
- *std_logic_arith*: định rõ các kiểu dữ liệu SIGNED và UNSIGNED, các giải thuật liên quan và so sánh toán tử. Nó cũng chứa nhiều hàm chuyển đổi dữ liệu, mà cho phép một kiểu được chuyển đổi thành các kiểu dữ liệu khác: *conv_integer(p),conv_unsigned(p, b), conv_signed(p, b), conv_std_logic_vector(p, b)*
- *std_logic_signed*: chứa các hàm cho phép làm việc với dữ liệu STD_LOGIC_VECTOR để được thực hiện chỉ khi dữ liệu là kiểu SIGNED
- *std_logic_signed*: chứa các hàm cho phép làm việc với dữ liệu STD_LOGIC_VECTOR để được thực hiện chỉ khi dữ liệu là kiểu UNSIGNED.

2.3. Entity (thực thể).

Một ENTITY là một danh sách mô tả các chân vào/ra (các PORT) của mạch điện. Cú pháp như sau:

```
ENTITY entity_name IS
PORT (
      port_name : signal_mode signal_type;
      port_name : signal_mode signal_type;
      ...);
END entity_name;
```

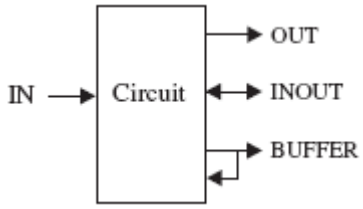
Chế độ của tín hiệu (mode of the signal) có thể là IN, OUT, INOUT hoặc BUFFER. Ví dụ trong hình 2.3 ta có thể thấy rõ các chân IN, OUT chỉ có một chiều (vào hoặc ra) trong khi INOUT là 2 chiều và BUFFER lại khác, tín hiệu ra phải được sử dụng từ dữ liệu bên trong.

Kiểu của tín hiệu (type of the signal) có thể là BIT, STD_LOGIC, INTEGER, …

Tên của thực thể (name of the entity) có thể lấy một tên bất kỳ, ngoại trừ các tù khóa của VHDL.

Ví dụ: Xét cổng NAND ở hình 2.4, khai báo ENTITY như sau:

```
ENTITY nand_gate IS
PORT (a, b : IN BIT;
x : OUT BIT);
END nand_gate;
```



Hình 2.3. Các chế độ tín hiệu

Hình 2.4. Cổng NAND

2.4. ARCHITECTURE (cấu trúc).

ARCHITECTURE là một mô tả mạch dùng để quyết mạch sẽ làm việc như thế nào (có chức năng gì).

Cú pháp như sau:

```
ARCHITECTURE architecture_name OF entity_name IS
[declarations]
BEGIN
(code)
END architecture_name;
```

Như thấy ở trên, một cấu trúc có 2 phần: phần khai báo (chức năng), nơi các tín hiệu và các hằng được khai báo, và phần mã (code - từ BEGIN trở xuống).

Ví dụ: Xét trở lại cổng NAND của hình 2.4

```
ARCHITECTURE myarch OF nand_gate IS
BEGIN
      x <= a NAND b;
END myarch;
```

Ý nghĩa của ARCHITECTURE trên là như sau: mạch phải thực hiện công việc NAND 2 tín hiệu vào (a,b) và gán (<=) kết quả cho chân ra x.
Mỗi một khai báo thực thể đều phải đi kèm với ít nhất một kiến trúc tương ứng. VHDL cho phép tạo ra hơn một kiến trúc cho một thực thể. Phần khai báo kiến trúc có thể bao gồm các khai báo về các tín hiệu bên trong, các phần tử bên trong hệ thống, hay các hàm và thủ tục mô tả hoạt động của hệ thống. Tên của kiến trúc là nhãn được đặt tuỳ theo người xử dụng. Có hai cách mô tả kiến trúc của một phần tử (hoặc hệ thống) đó là mô hình hoạt động (Behaviour) hay mô tả theo mô hình cấu trúc (Structure). Tuy nhiên một hệ thống có thể bao gồm cả mô tả theo mô hình hoạt động và mô tả theo mô hình cấu trúc.

+ ***Mô tả kiến trúc theo mô hình hoạt động:***

Mô hình hoạt động mô tả các hoạt động của hệ thống (hệ thống đáp ứng với các tín hiệu vào như thế nào và đưa ra kết quả gì ra đầu ra) dưới dạng các cấu trúc ngôn ngữ lập trình bậc cao. Cấu trúc đó có thể là PROCESS , WAIT, IF, CASE, FOR-LOOP…

Ví dụ:

```
ARCHITECTURE behavior OF nand IS
-- Khai báo các tín hiệu bên trong và các bí danh
BEGIN
      c <= NOT(a AND b);
END behavior;
```

Ví dụ2:

```
ARCHITECTURE behavioral of decode2x4 is
    BEGIN
```

```
Process (A,B,ENABLE)
      Variable ABAR,BBAR: bit;
Begin
           ABAR := not A;
           BBAR := not B;
       If ENABLE = '1' then
           Z(3) <= not (A and B);
           Z(0) <= not (ABAR and BBAR);
            Z(2)  <=  not (A and BBAR);
                Z(1)  <=  not (ABAR and B);
                   Else
      Z  <=  not (ABAR and B);
                    End if;
End process;
END arc_behavioral;
```

+ ***Mô tả kiến trúc theo mô hình cấu trúc:***

Mô hình cấu trúc của một phần tử (hoặc hệ thống) có thể bao gồm nhiều cấp cấu trúc bắt đầu từ một cổng logic đơn giản đến xây dựng mô tả cho một hệ thống hoàn thiện. Thực chất của việc mô tả theo mô hình cấu trúc là mô tả các phần tử con bên trong hệ thống và sự kết nối của các phần tử con đó.

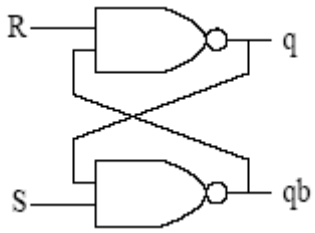
Mô tả cú pháp:

```
architecture identifier of entity_name is
                        Architecture_declarative_part
            begin
                 all_concurrent_statements
                        end
[architecture][architecture_simple_name];
```

Khai báo các thành phần:

```
Component
      Tên_componemt port [ danh sách ];
End component;
```

Như với ví dụ mô tả mô hình cấu trúc một flip-flop RS gồm hai cổng NAND có thể mô tả cổng NAND được định nghĩa tương tự như ví dụ với cổng NOT, sau đó mô tả sơ đồ móc nối các phần tử NAND tạo thành trigơ RS
Ví dụ1:



Hình 2.5.a. Sơ đồ của trigo RS

```
ENTITY rsff IS
PORT( r : IN std_logic;
      s : IN std_logic;
```

```
      q : OUT std_logic;
      qb : OUT std_logic);
END rsff;
ARCHITECTURE kien_truc OF rsff IS
COMPONENT nand        -- định nghĩa cổng nand
GENERIC(delay : time);
PORT(a : IN std_logic;
       b : IN std_logic;
      c : OUT std_logic);
END COMPONENT;
BEGIN
      u1: nand        -- cài đặt u1 là thành phần nand
      GENERIC MAP(5 ns) -- giá trị delay có thể thay đổi
      values
      PORT MAP(s, qb, q);   -- bản đồ I/O cho thành phần
      u2: nand        -- thiết lập u2 là thành phần nand
      GENERIC MAP(5 ns)
      PORT MAP(q, r, qb);
END kien_truc;
```

Ví dụ2:

```
Architecture  arc_mach_cong of mach_cong is
      Component  Xor
               Port( X,Y : in bit ; Z, T : out bit);
         End component;
         Component And
                     Port(L,M :input ;N,P : out bit );
         End component;
          Begin
                     G1 : Xor port map (A,B,Sum);
               G2 : And port map (A, B, C);
      End arc_mach_cong;
```

+ ***Mô tả kiến trúc theo mô hình tổng hợp***

Đó là mô hình kết hợp của 2 mô hình trên.

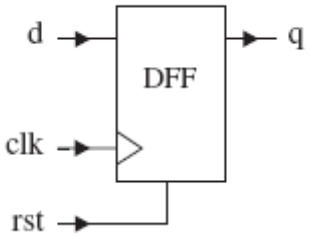
Ví dụ:

```
      Entity adder is
            Port (A,B,Ci : bit
                  S, Cout : bit);
      End  adder;

      Architecture arc_mixed of  adder is
            Component  Xor2
                  Port( P1, P2 : in bit;
                        PZ : out bit);
            End  compenent;
            Signal S1 :bit;
      Begin
            X1 : Xor2 port map(A,B,S1);
            Process (A,B,Cin)
                  Variable T1,T2,T3 : bit;
            Begin
```

```
                T1 := A and B;
                T2  :=  B and Cin ;
                T3 :=   A and Cin;
                Cout  :=  T1 or T2  or T3 ;
          End  process;
End arc_mixed ;
```

2.5. Các ví dụ mở đầu.

Trong mục này, chúng ta sẽ trình bày 2 ví dụ đầu tiên về mã VHDL. Mỗi ví dụ đều được theo kèm bởi các chú thích diễn giải và các kết quả mô phỏng.
Ví dụ 2.1: DFF với Reset không đồng bộ:



Hình 2.5.b. Sơ đồ của DFF không đồng bộ

Hình 2.5.b cho thấy sơ đồ của một flip-flop loại D (DFF), xung được kích theo sườn của tín hiệu đồng hồ (clk), và với một tín hiệu đầu vào reset không đồng bộ (rst). Khi rst = ‘1’, đầu ra luôn ở mức thấp bất kể clk. Ngược lại, đầu ra sẽ copy đầu vào (q<=d) tại thời điểm khi clk chuyển từ ‘0’ lên ‘1’.

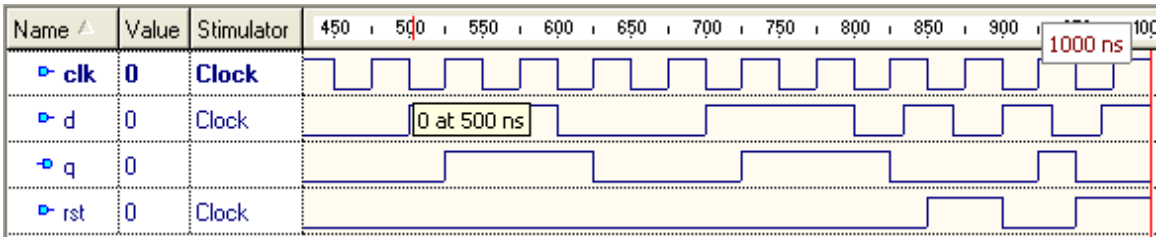
Có nhiều cách để thực hiện DFF của hình 2.5, một giải pháp sẽ được trình bày dưới đây. Sử dụng một PROCESS cho đoạn mã sau đây:

```
1    ---------------------------------------
2    LIBRARY ieee;
3    USE ieee.std_logic_1164.all;
4    ---------------------------------------
5    ENTITY dff IS
6         PORT ( d, clk, rst: IN STD_LOGIC;
7                q: OUT STD_LOGIC);
8    END dff;
9    ---------------------------------------
10   ARCHITECTURE behavior OF dff IS
11   BEGIN
12       PROCESS (rst, clk)
13       BEGIN
14         IF (rst='1') THEN
15             q <= '0';
16         ELSIF (clk'EVENT AND clk='1') THEN
17             q <= d;
18         END IF;
19        END PROCESS;
20   END behavior;
```

```
21    ------------------------------------
```

(Chú ý: VHDL không phân biệt chữ hoa và chữ thường.)

* Kết quả mô phỏng:

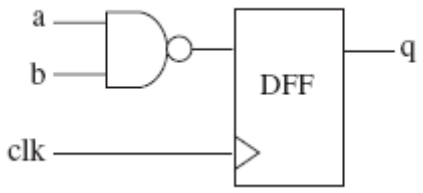


Hình 2.6: Kết quả mô phỏng của ví dụ 2.1

Hình 2.6 mô phỏng kết quả từ ví dụ 2.1, đồ thị có thể được giải thích dễ dàng. Cột đầu tiên cho biết tên của tín hiệu, như đã được đinh nghĩa trong ENTITY. Nó cũng cho biết chế độ (hướng) của tín hiệu, lưu ý rằng các mũi tên ứng với rst, d và clk hướng vào trong, đây là phía input, còn q hướng ra ngoài tương ứng với phía output. Cột thứ hai chứa giá trị của mỗi tín hiệu ở vị trí tương ứng với nơi con trỏ trỏ tới. Trong trường hợp hiện tại, con trỏ ở 0ns và tín hiệu nhận giá trị (1,0,0,0). Cột thứ 3 cho thấy sự mô phỏng của toàn bộ quá trình. Các tín hiệu vào (rst, d, clk) có thể được chọn một cách tự do và bộ mô phỏng sẽ xác định tín hiệu ngõ ra tương ứng.

Ví dụ 2.2: DFF kết hợp với cổng NAND

Mạch điện ở hình 2.7 là sự kết hợp của 2 hình 2.4 và 2.5. Trong lời giải sau đây, chúng ta đã giới thiệu một cách có chủ định một tín hiệu không cần thiết (temp), chỉ để minh họa một tín hiệu sẽ được khai báo như thế nào.



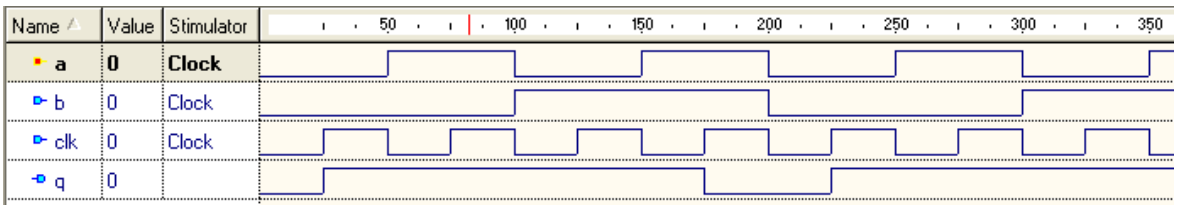
Hình 2.7. DFF kết hợp với cổng NAND

Mã thiết kế:

```
-------------------------------------
ENTITY example IS
PORT ( a, b, clk: IN BIT;
      q: OUT BIT);
END example;
-------------------------------------
ARCHITECTURE example OF example IS
SIGNAL temp : BIT;
BEGIN
      temp <= a NAND b;
      PROCESS (clk)
      BEGIN
```

```
            IF (clk'EVENT AND clk='1') THEN q<=temp;
            END IF;
      END PROCESS;
END example;
---------------------------------------
```

Kết quả mô phỏng từ mạch DFF kết hợp với NANDtrên hình 2.8:



Hình 2.8. Kết quả mô phỏng của ví dụ 2.2

Chương 3: Kiểu dữ liệu

Để viết mã VHDL một cách hiệu quả, thật cần thiết để biết rằng các kiểu dữ liệu nào được cho phép, làm thế nào để định rõ và sử dụng chúng. Trong chương này, tất cả các kiểu dữ liệu cơ bản sẽ được mô tả.

3.1. Các kiểu dữ liệu tiền định nghĩa.

VHDL bao gồm một nhóm các kiểu dữ liệu tiền định nghĩa, được định rõ thông qua các chuẩn IEEE 1076 và IEEE 1164. Cụ thể hơn, việc định nghĩa kiểu dữ liệu như thế có thể tìm thấy trong các gói/ thư viện sau:

- *Gói standard của thư viện std*: Định nghĩa các kiểu dữ liệu BIT, BOOLEAN, INTEGER và REAL.
- *Gói std_logic_1164 của thư viện ieee*: Định nghĩa kiểu dữ liệu STD_LOGIC và STD_ULOGIC.
- *Gói std_logic_arith của thư viện ieee*: Định nghĩa SIGNED và UNSIGNED, cộng thêm nhiều hàm chuyển đổi dữ liệu ví dụ: *conv_integer(p), conv_unsigned(p, b), conv_signed(p, b), và conv_std_logic_vector(p, b).*
- *Gói std_logic_signed và std_logic_unsigned của thư viện ieee*: Chứa các hàm cho phép hoạt động với dữ liệu STD_LOGIC_VECTOR được thực hiện khi mà kiểu dữ liệu là SIGNED hoặc UNSIGNED.

Tất cả các kiểu dữ liệu tiền định nghĩa đã nêu trên được mô tả như sau:

\+ BIT và BIT_VECTOR: 2 mức logic (‘0’, ’1’).
Ví dụ:

```
SIGNAL x: BIT;
```

```
-- x được khai báo như một tín hiệu số kiểu BIT.
SIGNAL y: BIT_VECTOR (3 DOWNTO 0);
-- y là một vec tơ 4 bit, với bit bên trái nhất được gọi là MSB.
SIGNAL w: BIT_VECTOR (0 TO 7);
-- w là một véc tơ8  bit, phía bên phải nhất được gọi là MSB
```

Dựa vào các tín hiệu ở trên, các phép gán sau đây là hợp lệ (để gán một giá trị đến một tín hiệu, toán tử <= được sử dụng):

```
x <= "1";
y <= "0111";
z <= "01110001";
```

+ STD_LOGIC (và STD_LOGIC_VECTOR):

Hệ logic 8 giá trị sau đây được giới tiệu trong chuẩn IEEE 1164:

'X'	không xác định (bắt buộc)
'0'	mức thấp (bắt buộc)
'1'	mức cao (bắt buộc)
'Z'	trở kháng cao
'W'	không xác định (yếu)
'L'	mức thấp (yếu)
'H'	mức cao (yếu)
'-'	không quan tâm

Ví dụ:

```
SIGNAL x: STD_LOGIC;
-- x được khai báo như một ký tự số ( vô hướng), tín hiệu thuộc
kiểu STD_LOGIC
SIGNAL y: STD_LOGIC_VECTOR (3 DOWNTO 0) := "0001";
-- y được khai báo như một vector 4-bit, với bit bên trái cùng là
-- MSB. Giá trị khởi đầu của y là "0001". Lưu ý
-- rằng toán tử ":=" được sử dụng để thiết lập giá trị khởi đầu.
```

Hầu hết các mức std_logic là vô hướng chỉ đối với quá trình mô phỏng. Tuy nhiên '0', '1' và 'Z' là có thể kết hợp không hạn chế. Đối với các giá trị "weak", chúng được giải quyết trong sự ưu tiên của các giá trị "forcing" trong các nút đa chiều (Bảng 3.1). Thật vậy, nếu 2 tín hiệu std_logic bất kỳ được nối đến cùng một node, thì các mức logic đối lập được tự động giải quyết theo Bảng 3.1

	X	0	1	Z	W	L	H	-
X	X	X	X	X	X	X	X	X
0	X	0	X	0	0	0	0	X
1	X	X	1	1	1	1	1	X
Z	X	0	1	Z	W	L	H	X
W	X	0	1	W	W	W	W	X
L	X	0	1	L	W	L	W	X
H	X	0	1	H	W	W	H	X
-	X	X	X	X	X	X	X	X

Bảng 3.1. Hệ thống logic giải được

+ STD_ULOGIC(STD_ULOGIC_VECTOR): hệ thống logic 9 mức trong chuẩn IEEE 1164: (‘U’, ‘X’, ‘0’, ‘1’, ‘Z’, ‘W’, ‘L’, ‘H’, ‘–’). Thật vậy, hệ STD_LOGIC mô tả ở trên là một tập con của STD_ULOGIC. Hệ thống thứ 2 này thêm giá trị logic ‘U’.

- BOOLEAN: đúng/sai
- INTEGER: số nguyên 32 bits (từ -2.147.483.647 đến +2.147.483.647)
- NATURAL: msố nguyên không âm (từ 0 đến +2.147.483.647)
- REAL: số thực nằm trong khoảng (từ -1.0E38 đến +1.0E38).
- Physic literals: sử dụng đối với các đại lượng vật lý, như thời gian, điện áp,…Hữu ích trong mô phỏng
- Character literals: ký tự ASCII đơn hoặc một chuỗi các ký tự như thế
- SIGNED và UNSIGNED: các kiểu dữ liệu được định nghĩa trong gói *std_logic_arith* của thư viện *ieee*. Chúng có hình thức giống như STD_LOGIC_VECTOR, nhưng ngoại trừ các toán tử số học, mà tiêu biểu là kiểu dữ liệu INTEGER

Các ví dụ:

```
x0 <= '0';              -- bit, std_logic, or std_ulogic value '0'
x1 <= "00011111";       -- bit_vector, std_logic_vector,
                        -- std_ulogic_vector, signed, or unsigned
x2 <= "0001_1111"; -- đường gạch dưới cho phép dễ hình dung hơn
x3 <= "101111"          -- biểu diễn nhị phân của số thập phân 47
x4 <= B"101111"         -- như trên
x5 <= O"57"             -- biểu diễn bát phân của số thập phân 47
x6 <= X"2F"             -- biểu diễn số thập lục phân của số thập phân 47
n <= 1200;              -- số nguyên
m <= 1_200;             -- số nguyên, cho phép gạch dưới
IF ready THEN...        -- Logic, thực hiện nếu ready=TRUE
y <= 1.2E-5;            -- real, not synthesizable
q <= d after 10 ns;     -- physical, not synthesizable
```

Ví dụ: Các toán tử được phép và không được phép nằm giữa các kiểu dữ liệu khác nhau:

```
SIGNAL a: BIT;
SIGNAL b: BIT_VECTOR(7 DOWNTO 0);
SIGNAL c: STD_LOGIC;
SIGNAL d: STD_LOGIC_VECTOR(7 DOWNTO 0);
SIGNAL e: INTEGER RANGE 0 TO 255;
...
a <= b(5); -- được phép (cùng kiểu vô hướng: BIT)
b(0) <= a; -- được phép (cùng kiểu vô hướng: BIT)
c <= d(5); -- được phép (cùng kiểu vô hướng: STD_LOGIC)
d(0) <= c; -- được phép (cùng kiểu vô hướng: STD_LOGIC)
a <= c; -- không được phép (không thể kết hợp kiểu: BIT x STD_LOGIC)
b <= d; -- không được phép (không thể kết hợp kiểu: BIT_VECTOR x
        -- STD_LOGIC_VECTOR)
e <= b; -- không được phép (không thể kết hợp kiểu: INTEGER x BIT_VECTOR)
e <= d; -- không được phép (không thể kết hợp kiểu: INTEGER x
        -- STD_LOGIC_VECTOR)
```

3.2. Các kiểu dữ liệu người dùng định nghĩa.

VHDL cũng cho phép người dùng tự định nghĩa các kiểu dữ liệu. Hai loại kiểu dữ liệu người dùng định nghĩa được chỉ ra dưới đây bao gồm integer và enumerated.

Kiểu integer người dùng định nghĩa:

```
TYPE integer IS RANGE -2147483647 TO +2147483647;
-- Thực ra kiểu này đã được định nghĩa trước bởi kiểu INTEGER.
TYPE natural IS RANGE 0 TO +2147483647;
-- Thực ra kiểu này được đã định nghĩa trước bởi kiểu NATURAL.
TYPE my_integer IS RANGE -32 TO 32;
-- Một tập con các số integer mà người dùng định nghĩa.
TYPE student_grade IS RANGE 0 TO 100;
-- Một tập con các số nguyên hoặc số tự nhiên người dùng định nghĩa.
_ Các kiểu đếm người dùng đinh nghĩa:
TYPE bit IS ('0', '1');
-- Được định nghĩa trước bởi kiểu BIT
TYPE my_logic IS ('0', '1', 'Z');
```

-- Một tập con của std_logic mà người dùng định nghĩa
TYPE bit_vector IS ARRAY (NATURAL RANGE <>) OF BIT;
-- đã được định nghĩa trước bởi BIT_VECTOR.
-- RANGE <> được sủ dụng để chỉ thị rằng các mức.không giới hạn.
-- NATURAL RANGE <>, on the other hand, indicates that the only
-- restriction is that the range must fall within the NATURAL
-- range.
TYPE state IS (idle, forward, backward, stop);
-- Một kiểu dữ liệu , điển hình của các máy trạng thái hữu hạn.
TYPE color IS (red, green, blue, white);
-- Kiểu dữ liệu liệt kê khác.

Việc mã hóa các kiểu liệt kê được thực hiện một cách tuần tự và tự động.

Ví dụ: Cho kiểu màu như ở trên, để mã hóa cần 2 bit (có 4 trạng thái), bắt đầu ’00’ được gán cho trạng thái đầu tiên (red), ‘01’ được gán cho trạng thái thứ hai (green), ‘10’ kế tiếp (blue) và cuối cùng là trạng thái ‘11’ (while).

3.3. Các kiểu con (Subtypes).

Kiểu dữ liệu con là một kiểu dữ liệu đi kèm theo điều kiện ràng buộc. Lý do chính cho việc sử dụng kiểu dữ liệu con để sau đó định ra một kiểu dữ liệu mới đó là, các thao tác giữa các kiểu dữ liệu khác nhau không được cho phép, chúng chỉ được cho phép trong trường hợp giữa một kiểu con và kiểu cơ sở tương ứng với nó.

Ví dụ: kiểu dữ liệu sau đây nhận được các kiểu dữ liệu được giới thiệu trong các ví dụ phần trước.

SUBTYPE natural IS INTEGER RANGE 0 TO INTEGER'HIGH;
-- NATURAL is a kiểu con (tập con) of INTEGER.
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO 'Z';
-- Gọi lại STD_LOGIC=('X','0','1','Z','W','L','H','-').
-- Do đó, my_logic=('0','1','Z').
SUBTYPE my_color IS color RANGE red TO blue;
-- khi color=(red, green, blue, white), thì
-- my_color=(red, green, blue).
SUBTYPE small_integer IS INTEGER RANGE -32 TO 32;
-- Một tập con của INTEGER.
Example: Các phép toán hợp lệ và không hợp lệ giữa các kiểu dữ liệu và các kiểu dữ liệu con.
SUBTYPE my_logic IS STD_LOGIC RANGE '0' TO '1';
SIGNAL a: BIT;
SIGNAL b: STD_LOGIC;
SIGNAL c: my_logic;

...
b <= a; --không hợp lệ (không thể kết hợp kiểu: BIT với STD_LOGIC)
b <= c; --hợp lệ (cùng kiểu cơ sở: STD_LOGIC)

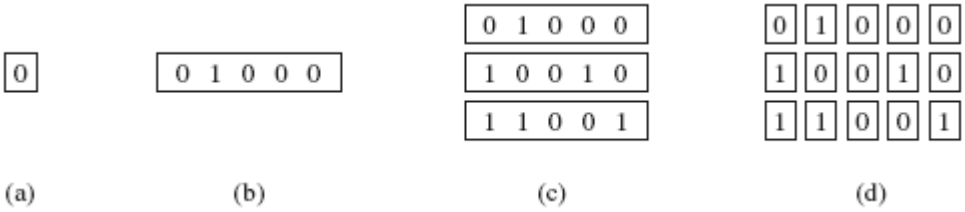
3.4. Mảng (Arrays).

Mảng là một tập hợp các đối tượng có cùng kiểu. Chúng có thể là một chiều (1D), 2 chiều (2D) hoặc một chiều của một chiều (1D x 1D) và cũng có thể có những kích thước cao hơn.

Hình 3.1 minh họa việc xây dựng một mảng dữ liệu. Một giá trị đơn (vô hướng được chỉ ra ở (a), một vector (mảng 1D) ở (b) và một mảng các vector (mảng 1Dx1D) ở (c) và mảng của mảng 2D như trong (d)

Thật vậy, các kiểu dữ liệu VHDL được định nghĩa trước đó (mục 3.1) chỉ bao gồm các đại lượng vô hướng-scalar (bit đơn) và vector (mảng một chiều các bit). Các kiểu dữ liệu có thể kết hợp trong mỗi loại này là như dưới đây:

_ Scalars: BIT, STD_LOGIC, STD_ULOGIC, and BOOLEAN.
_ Vectors: BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR,
INTEGER, SIGNED, and UNSIGNED.



Hình 3.1: Minh họa scalar (a), 1D (b), 1Dx1D (c), và 2D (d)

Như có thể thấy, không hề có định nghĩa trước mảng 2D hoặc 1Dx1D, mà khi cần thiết, cần phải được chỉ định bởi người dùng. Để làm như vậy, một kiểu mới (new TYPE) cần phải được định nghĩa đầu tiên, sau đó là tín hiệu mới (new SIGNAL), new VARIABLE hoặc CONSTANT có thể được khai báo sử dụng kiểu dữ liệu đó. Cú pháp dưới đây sẽ được dùng:

Để chỉ định một kiểu mảng mới:

TYPE type_name IS ARRAY (specification) OF data_type;

Để tạo sử dụng kiểu mảng mới:

SIGNAL signal_name: type_name [:= initial_value];

Trong cú pháp ở trên, một SIGNAL được khai báo. Tuy nhiên nó cũng có thể là một CONSTANT hoặc một VARIABLE. Gia trị khởi tạo tùy chọn.

** Ví dụ mảng 1Dx1D*:

Chúng ta muốn xây dựng một mảng chứa 4 vector, mỗi vector có kích thước là 8 bit, đólà một mảng 1Dx1D (hình 3.1). Ta gọi mỗi vector là hàng *(row)* và mảng hoàn chỉnh là ma trận *(matrix)*. Hơn nữa, chúng ta muốn bit bên

trái cùng của mỗi vector trở thành MSB (most significant bit) của nó, và dòng trên cùng trở thành dòng 0. Khi đó sự thực hiện đầy đủ mảng sẽ là như sau:

```
TYPE row IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D array
TYPE matrix IS ARRAY (0 TO 3) OF row; -- 1Dx1D array
SIGNAL x: matrix; -- 1Dx1D signal
```

** Ví dụ mảng 1Dx1D khác:*

Cách khác để xây dựng mảng 1Dx1D ở trên còn được thực hiện như sau:

```
TYPE matrix IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7 DOWNTO 0);
```

** Ví dụ mảng 2D:*

Mảng sau đây thực sự là hai chiều. Lưu ý rằng việc xây dựng nó dựa trên các vector, nhưng khá hoàn chỉnh trên các đại lượng vô hướng.

```
TYPE matrix2D IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
-- 2D array
```

** Khởi đầu cho mảng:*

Như đã thấy trong cú pháp ở trên, giá trị khởi đầu của một SIGNAL hoặc VARIABLE là tùy chọn. Tuy nhiên, khi việc khởi đầu giá trị được đòi hỏi, nó có thể được thực hiện như trong ví dụ phía dưới đây:

```
... :="0001";                          -- for 1D array
... :=('0','0','0','1')                -- for 1D array
... :=(('0','1','1','1'), ('1','1','1','0')); -- for 1Dx1D or-- 2D array
```

** Ví dụ: Các phép gán mảng hợp lệ và không hợp lệ*

Phép gán trong ví dụ này được dựa trên định nghĩa kiểu và khai báo các tín hiệu như sau:

```
TYPE row IS ARRAY (7 DOWNTO 0)OF STD_LOGIC;
                                        -- 1D array
TYPE array1 IS ARRAY (0 TO 3) OF row;
                                        -- 1Dx1D array
TYPE array2 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(7
DOWNTO 0);
                                        -- 1Dx1D
TYPE array3 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC;
                                        -- 2D array
SIGNAL x: row;
SIGNAL y: array1;
SIGNAL v: array2;
SIGNAL w: array3;
```

--------- Các phép gán vô hướng hợp lệ: ----------
-- Các phép gán đại lượng vô hướng (bit đơn) dưới đây là hợp lệ,
-- bởi vì kiểu (vô hướng) cơ bản là STD_LOGIC cho tất cả các tín hiệu
-- (x,y,v,w).

```
x(0) <= y(1)(2); -- lưu ý 2 cặp dấu ngoặc đơn
            -- (y is 1Dx1D)
x(1) <= v(2)(3); -- 2 cặp dấu  ngoặc đơn (v is 1Dx1D)
x(2) <= w(2,1); -- 1 cặp dấu  ngoặc đơn (w is 2D)
y(1)(1) <= x(6);
y(2)(0) <= v(0)(0);
y(0)(0) <= w(3,3);
w(1,1) <= x(7);
w(3,0) <= v(0)(3);
```

--------- Gán vector: ----------

```
x <= y(0);          -- hợp lệ (cùng kiểu: ROW)
x <= v(1);          -- không hợp lệ (không phù hợp kiểu: ROW và
                    -- STD_LOGIC_VECTOR)
x <= w(2);          -- không hợp lệ (w phải là 2D)
x <= w(2,2 DOWNTO 0);--không hợp lệ (không phù hợp kiểu: ROW x
                          -- STD_LOGIC)
v(0)<=w(2,2 DOWNTO 0);--illegal(mismatch: STD_LOGIC_VECTOR
                          -- x STD_LOGIC)
v(0) <= w(2);             -- illegal (w must have 2D index)
y(1) <= v(3);                   -- illegal (type mismatch: ROW x
                          -- STD_LOGIC_VECTOR)
y(1)(7 DOWNTO 3) <= x(4 DOWNTO 0); -- legal (same type,
                                   -- same size)
v(1)(7 DOWNTO 3) <= v(2)(4 DOWNTO 0); -- legal (same type,
                                   -- same size)
w(1,5 DOWNTO 1)<=v(2)(4 DOWNTO 0);-- illegal (type      mismatch)
```

3.5. Mảng cổng (Port Array).

Như chúng ta đã biết, không có kiểu dữ liệu được định nghĩa trước nào có hơn một chiều. Tuy nhiên, trong các đặc điểm của các chân vào hoặc ra (các PORT) của một mạch điện (mà được xây dựng thành ENTITY), chúng ta có thể phải cần định rõ các PORT như là mảng các VECTOR

Khi các khai báo TYPE không được cho phép trong một ENTITY, giải pháp để khai báo kiểu dữ liệu người dùng định nghĩa trong một PACKAGE, mà có thể nhận biết toàn bộ thiết kế. Một ví dụ như sau:

```
------- Package: --------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------
PACKAGE my_data_types IS
TYPE vector_array IS ARRAY (NATURAL RANGE <>) OF
STD_LOGIC_VECTOR(7 DOWNTO 0);
```

```
END my_data_types;
-------------------------------------------
------- Main code: ------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE work.my_data_types.all; -- user-defined package
---------------------------
ENTITY mux IS
PORT (inp: IN VECTOR_ARRAY (0 TO 3);
... );
END mux;
... ;
-------------------------------------------
```

Có thể thấy trong ví dụ trên, một kiểu dữ liệu người dùng định nghĩa được gọi là *vector_array*, đã được tạo ra, mà nó có thể chứa một số không xác định các vector, mỗi vector chứa 8 bit. Kiểu dữ liệu được lưu giữ trong một PACKAGE gọi là my_data_types, và sau đó được sử dụng trong một ENTITY để xác định một PORT được gọi. Chú ý trong đoạn mã chính bao gồm thêm cả một mệnh đề USE để thực hiện gói người dùng định nghĩa *my_data_types* có thể thấy trong thiết kế.

Chức năng khác cho PACKAGE ở trên sẽ được trình bày dưới đây, nơi mà có khai báo CONSTANT:

```
-------                                    Package:
------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------
PACKAGE my_data_types IS
CONSTANT b: INTEGER := 7;
TYPE vector_array IS ARRAY (NATURAL RANGE <>)
OF
STD_LOGIC_VECTOR(b DOWNTO 0);
END my_data_types;
-----------------------------------------------
```

3.6. Kiểu bản ghi (Records).

Bản ghi tương tự như mảng, với điểm khác rằng chúng chứa các đối tượng có kiểu dữ liệu khác nhau.
Ví dụ:

```
TYPE birthday IS RECORD
day: INTEGER RANGE 1 TO 31;
month: month_name;
END RECORD;
```

3.7. Kiểu dữ liệu có dấu và không dấu (Signed and Unsigned).

Như đã đề cập trước đây, các kiểu dữ liệu này được định nghĩa trong gói *std_logic_arith* của thư viện *ieee*. Cú pháp của chúng được minh họa trong ví dụ dưới đây:

Ví dụ:

```
SIGNAL x: SIGNED (7 DOWNTO 0);
SIGNAL y: UNSIGNED (0 TO 3);
```

Lưu ý rằng cú pháp của chúng tương tự với STD_LOGIC_VECTOR, không giống như INTEGER.

Một giá trị UNSIGNED là một số không bao giờ nhỏ hơn zero. Ví dụ, "0101" biểu diễn số thập phân 5, trong khi "1101" là 13. Nhưng nếu kiểu SIGNED được sử dụng thay vào, giá trị có thể là dương hoặc âm (theo định dạng bù 2). Do đó, "0101" vẫn biểu diễn số 5, trong khi "1101" sẽ biểu diễn số -3

Để sử dụng kiểu dữ liệu SIGNED hoặc UNSIGNED, gói *std_logic_arith* của thư viện *ieee*, phải được khai báo. Bất chấp cú pháp của chúng, kiểu dữ liệu SIGNED và UNSIGNED có hiệu quả chủ yếu đối với các phép toán số học, nghĩa là, ngược với STD_LOGIC_VECTOR, chúng chấp nhận các phép toán số học. Ở một khía cạnh khác, các phép toán logic thì không được phép.

* Ví dụ:

Các phép toán hợp lệ và không hợp lệ đối với kiểu dữ liệu signed/unsigned:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;    -- gói cần thiết
thêm vào
...
SIGNAL a: IN SIGNED (7 DOWNTO 0);
SIGNAL b: IN SIGNED (7 DOWNTO 0);
SIGNAL x: OUT SIGNED (7 DOWNTO 0);
...
v <= a + b;           -- hợp lệ (phép toán số học
OK)
w <= a AND b;    -- không hợp lệ (phép toán logic
không OK)
```

Các phép toán hợp lệ và không hợp lệ với std_logic_vector:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all; -- không thêm gói đòi
hỏi
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
```

```
v <= a + b;                -- không hợp lệ (phép
toán số học không OK)
w <= a AND b;          -- hợp lệ (phép toán logic OK)
```

* Ví dụ: Các phép toán số học với *std_logic_vector*

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all; -- bao gồm gói thêm
vào
...
SIGNAL a: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
SIGNAL x: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
v <= a + b;            -- hợp lệ (phép toán số học
OK), không dấu
w <= a AND b;    -- hợp lệ (phép toán logic OK).
```

3.8. Chuyển đổi dữ liệu.

VHDL không cho phép các phép toán trực tiếp (số học, logic, …) tác động lên các dữ liệu khác kiểu nhau. Do đó, thường là rất cần thiết đối với việc chuyển đổi dữ liệu từ một kiểu này sang một kiểu khác. Điều này có thể được thực hiện trong hai cách cơ bản: hoặc chúng ta viết một ít code cho điều đó, hoặc chúng ta gọi một FUNCTION từ một gói được định nghĩa trước mà nó cho phép thực hiện các phép biến đổi cho ta.

Nếu dữ liệu được quan hệ đóng (nghĩa là 2 toán hạng có cùng kiểu cơ sở, bất chấp đang được khai báo thuộc về hai kiểu lớp khác nhau), thì std_logic_1164 của thư viện ieee cung cấp các hàm chuyển đổi dễ thực hiện.

* Ví dụ: các phép toán hợp lệ và không hợp lệ đối với các tập con

```
TYPE long IS INTEGER RANGE -100 TO 100;
TYPE short IS INTEGER RANGE -10 TO 10;
SIGNAL x : short;
SIGNAL y : long;
...
y <= 2*x + 5;          -- lỗi, không phù hợp kiểu
y <= long(2*x + 5);    -- OK, kết quả được chuyển đổi
thành kiểu long
```

Nhiều hàm chuyển đổi dữ liệu có thể được tìm trong gói std_logic_arith của thư viện ieee:

- ***conv_integer(p)***: chuyển đổi một tham số p của kiểu INTEGER, UNSIGNED, SIGNED, hoặc STD_ULOGIC thành một giá trị INTEGER. Lưu ý rằng STD_LOGIC_VECTOR không được kể đến.

- ***conv_unsigned(p, b)***: chuyển đổi một tham số p của kiểu INTEGER, UNSIGNED, SIGNED, hoặc STD_ULOGIC thành một giá trị UNSIGNED với kích cỡ là b bit.
- ***conv_signed(p, b):*** chuyển đổi một tham số p của kiểu INTEGER, UNSIGNED, SIGNED, hoặc STD_ULOGIC thành một giá trị SIGNED với kích cỡ là b bits.
- ***conv_std_logic_vector(p, b):*** chuyển đổi một tham số p thuộc kiểu dữ liệu INTEGER, UNSIGNED, SIGNED, hoặc STD_LOGIC thành một giá trị STD_LOGIC_VECTOR với kích thước b bits.

* Ví dụ: chuyển đổi dữ liệu:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
...
SIGNAL a: IN UNSIGNED (7 DOWNTO 0);
SIGNAL b: IN UNSIGNED (7 DOWNTO 0);
SIGNAL y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0);
...
y <= CONV_STD_LOGIC_VECTOR ((a+b), 8);
-- Phép toán hợp lệ: a+b được chuyển đổi từ UNSIGNED
thành một
-- giá trị 8-bit STD_LOGIC_VECTOR, sau đó gán cho y.
```

Một cách khác có thể chọn đã được đề cập đến trong mục trước đây. Nó bao gồm việc sử dụng các gói *std_logic_signed* và *std_logic_unsigned* từ thư viện *ieee.* Các gói này cho phép các phép toán với dữ liệu STD_LOGIC_VECTOR được thực hiện nếu dữ liệu đã là kiểu SIGNED hoặc UNSIGNED, một cách lần lượt.

3.9. Tóm tắt.

Các kiểu dữ liệu VHDL tổng hợp cơ bản được tóm tắt trong bảng 3.2

Data types	Synthesizable values
BIT, BIT_VECTOR	'0', '1'
STD_LOGIC, STD_LOGIC_VECTOR	'X', '0', '1', 'Z' (resolved)
STD_ULOGIC, STD_ULOGIC_VECTOR	'X', '0', '1', 'Z' (unresolved)
BOOLEAN	True, False
NATURAL	From 0 to +2, 147, 483, 647
INTEGER	From −2,147,483,647 to +2,147,483,647
SIGNED	From −2,147,483,647 to +2,147,483,647
UNSIGNED	From 0 to +2,147,483,647
User-defined integer type	Subset of INTEGER
User-defined enumerated type	Collection enumerated by user
SUBTYPE	Subset of any type (pre- or user-defined)
ARRAY	Single-type collection of any type above
RECORD	Multiple-type collection of any types above

Bảng 3.2. Tổng hợp các kiểu dữ liệu.

3.10. Các ví dụ.

* Ví dụ 3.1: Sự phân chia đối với các kiểu dữ liệu

Các phép gán hợp lệ và không hợp lệ được trình bày kế tiếp được dựa trên các định nghĩa kiểu và các khai báo tín hiệu sau đây:

```
TYPE byte IS ARRAY (7 DOWNTO 0) OF STD_LOGIC; -- 1D
-- array
TYPE mem1 IS ARRAY (0 TO 3, 7 DOWNTO 0) OF STD_LOGIC; -- 2D
-- array
TYPE mem2 IS ARRAY (0 TO 3) OF byte; -- 1Dx1D
-- array
TYPE mem3 IS ARRAY (0 TO 3) OF STD_LOGIC_VECTOR(0 TO 7); -- 1Dx1D
-- array
SIGNAL a: STD_LOGIC; -- scalar signal
SIGNAL b: BIT; -- scalar signal
SIGNAL x: byte; -- 1D signal
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0); -- 1D signal
SIGNAL v: BIT_VECTOR (3 DOWNTO 0); -- 1D signal
SIGNAL z: STD_LOGIC_VECTOR (x'HIGH DOWNTO 0); -- 1D signal
SIGNAL w1: mem1; -- 2D signal
SIGNAL w2: mem2; -- 1Dx1D signal
SIGNAL w3: mem3; -- 1Dx1D signal
-------- Legal scalar assignments: ----------------------
x(2) <= a; -- same types (STD_LOGIC), correct indexing
y(0) <= x(0); -- same types (STD_LOGIC), correct indexing
z(7) <= x(5); -- same types (STD_LOGIC), correct indexing
b <= v(3); -- same types (BIT), correct indexing
w1(0,0) <= x(3); -- same types (STD_LOGIC), correct indexing
```

Table 3.2
Synthesizable data types.
Data types Synthesizable values
BIT, BIT_VECTOR '0', '1'

STD_LOGIC, STD_LOGIC_VECTOR ‘X’, ‘0’, ‘1’, ‘Z’ (resolved)
STD_ULOGIC, STD_ULOGIC_VECTOR ‘X’, ‘0’, ‘1’, ‘Z’ (unresolved)
BOOLEAN True, False
NATURAL From 0 to þ2, 147, 483, 647
INTEGER From _2,147,483,647 to þ2,147,483,647
SIGNED From _2,147,483,647 to þ2,147,483,647
UNSIGNED From 0 to þ2,147,483,647
User-defined integer type Subset of INTEGER
User-defined enumerated type Collection enumerated by user
SUBTYPE Subset of any type (pre- or user-defined)
ARRAY Single-type collection of any type above
RECORD Multiple-type collection of any types above
Data Types 39
TLFeBOOK
w1(2,5) <= y(7); -- same types (STD_LOGIC), correct indexing
w2(0)(0) <= x(2); -- same types (STD_LOGIC), correct indexing
w2(2)(5) <= y(7); -- same types (STD_LOGIC), correct indexing
w1(2,5) <= w2(3)(7); -- same types (STD_LOGIC), correct indexing
------- Illegal scalar assignments: --------------------
b <= a; -- type mismatch (BIT x STD_LOGIC)
w1(0)(2) <= x(2); -- index of w1 must be 2D
w2(2,0) <= a; -- index of w2 must be 1Dx1D
------- Legal vector assignments: ----------------------
x <= "11111110";
y <= ('1','1','1','1','1','1','0','Z');
z <= "11111" & "000";
x <= (OTHERS => '1');
y <= (7 =>'0', 1 =>'0', OTHERS => '1');
z <= y;
y(2 DOWNTO 0) <= z(6 DOWNTO 4);
w2(0)(7 DOWNTO 0) <= "11110000";
w3(2) <= y;
z <= w3(1);
z(5 DOWNTO 0) <= w3(1)(2 TO 7);
w3(1) <= "00000000";
w3(1) <= (OTHERS => '0');
w2 <= ((OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'),(OTHERS=>'0'));
w3 <= ("11111100", ('0','0','0','0','Z','Z','Z','Z',),
(OTHERS=>'0'), (OTHERS=>'0'));
w1 <= ((OTHERS=>'Z'), "11110000" ,"11110000", (OTHERS=>'0'));
------ Illegal array assignments: ----------------------
x <= y; -- type mismatch
y(5 TO 7) <= z(6 DOWNTO 0); -- wrong direction of y
w1 <= (OTHERS => '1'); -- w1 is a 2D array
w1(0, 7 DOWNTO 0) <="11111111"; -- w1 is a 2D array
w2 <= (OTHERS => 'Z'); -- w2 is a 1Dx1D array
w2(0, 7 DOWNTO 0) <= "11110000"; -- index should be 1Dx1D
-- Example of data type independent array initialization:
FOR i IN 0 TO 3 LOOP

```
FOR j IN 7 DOWNTO 0 LOOP
x(j) <= '0';
y(j) <= '0'
40 Chapter 3
TLFeBOOK
z(j) <= '0';
w1(i,j) <= '0';
w2(i)(j) <= '0';
w3(i)(j) <= '0';
END LOOP;
END LOOP;
```

---

* Ví dụ 3.2: Bit đơn và bit vector

Ví dụ này minh họa sự khác nhau giữa phép gán một bit đơn và phép gán một bit vector (nghĩa là, BIT với BIT_VECTOR, STD_LOGIC với STD_LOGIC_VECTOR, hoặc STD_ULOGIC với STD_ULOGIC_VECTOR).

Hai đoạn mã VHDL được giới thiệu phía dưới. Cả hai thực hiện phép toán AND giữa các tín hiệu vào và gán kết quả đến tín hiệu ra. Chỉ có một sự khác biệt giữa chúng đó là số lượng bit ở cổng vào và cổng ra (một bit trong ví dụ đấu tiên, 4 bits trong ví dụ thứ hai). Mạch điện suy ra từ các đoạn mã này được biểu diễn trên hình 3.2:

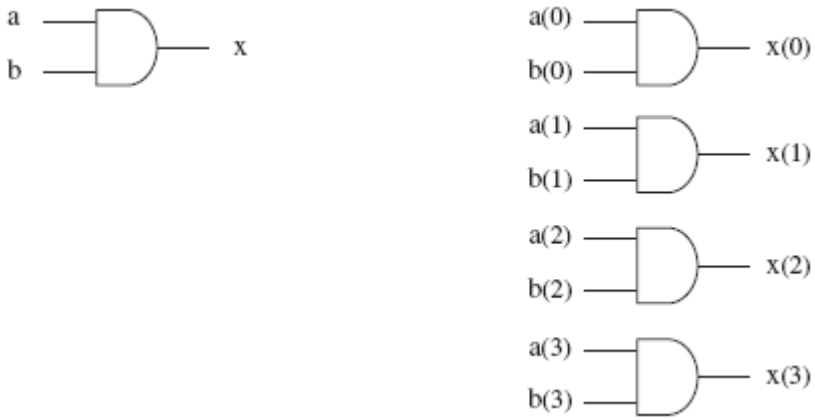
```
-- code 1----------
```

```
ENTITY and2 IS
        PORT (a, b: IN BIT;
                x: OUT BIT);
END and2;
```

```
ARCHITECTURE and2 OF and2 IS
BEGIN
        x <= a AND b;
END and2;
```

```
----code 2----------
ENTITY and2 IS
        PORT (a, b: IN BIT_VECTOR (0 TO 3);
              x: OUT BIT_VECTOR (0 TO 3));
END and2;
```

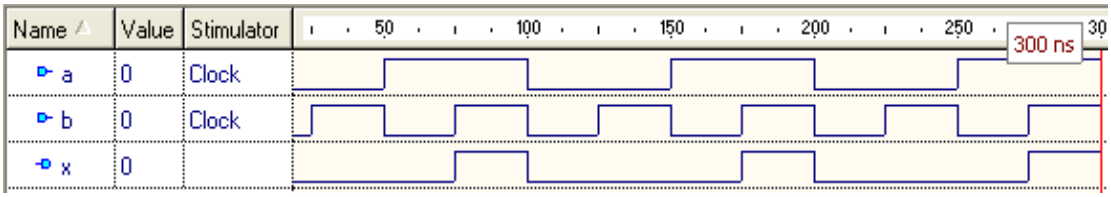
```
ARCHITECTURE and2 OF and2 IS
BEGIN
        x <= a AND b;
END and2
```



Hình 3.2. M ạch được suy ra từ mã của v í dụ 3.2

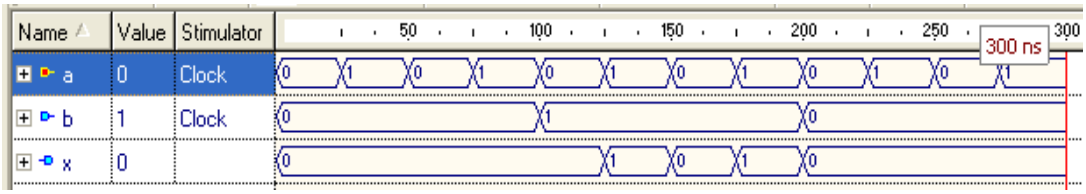
Kết quả mô phỏng trên Active HDL 6.1:

Code 1:



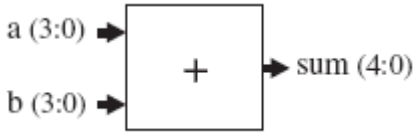
Hình 3.2.a. Kết quả mô phỏng cho đoạn mã 1của ví dụ 3.2

Code 2:



Hình 3.2.b. Kết quả mô phỏng cho đoạn mã 1của ví dụ 3.2

* Ví dụ 3.3: Bộ cộng (Adder)



Hình 3.3. Bộ cộng 4 bit cho ví dụ 3.3

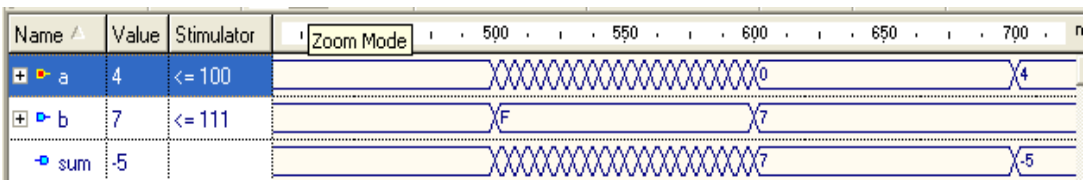
Hình 3.3 cho thấy giản đồ mức đỉnh của một bộ cộng 4 bit, mạch điện có 2 đầu vào (a,b) và một đầu ra (sum). Có 2 giải pháp được đề cập. Thứ nhất, tất cả các tín hiệu có kiểu dữ liệu SIGNED, trong khi ở giải pháp thứ hai đầu ra có kiểu INTEGER. Lưu ý trong giải pháp thứ hai có một hàm chuyển đổi

(conversion function) được sử dụng ở dòng 13, để kiểu của (a+b) phù hợp với kiểu của tổng. Lưu ý cần bao gồm cả gói std_logic_arith (dòng 4 của mỗi giải pháp), có mô tả kiểu dữ liệu SIGNED. Nhớ lại rằng một giá trị SIGNED được mô tả giống như một vector, nghĩa là, tương tự như STD_LOGIC_VECTOR, không giống INTEGER.

Code:

```
-------------------------------------------------
1     ----- Solution 1: in/out=SIGNED ----------
2     LIBRARY ieee;
3     USE ieee.std_logic_1164.all;
4     USE ieee.std_logic_arith.all;
5     ----------------------------------------
6     ENTITY adder1 IS
7           PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8           sum : OUT SIGNED (4 DOWNTO 0));
9     END adder1;
10    ----------------------------------------
11    ARCHITECTURE adder1 OF adder1 IS
12    BEGIN
13          sum <= a + b;
14    END adder1;
15    ----------------------------------------
-------------------------------------------------
1     ------ Solution 2: out=INTEGER -----------
2     LIBRARY ieee;
3     USE ieee.std_logic_1164.all;
4     USE ieee.std_logic_arith.all;
5     ----------------------------------------
6     ENTITY adder2 IS
7           PORT ( a, b : IN SIGNED (3 DOWNTO 0);
8           sum : OUT INTEGER RANGE -16 TO 15);
9     END adder2;
10    ----------------------------------------
11    ARCHITECTURE adder2 OF adder2 IS
12    BEGIN
13          sum <= CONV_INTEGER(a + b);
14    END adder2;
15    ----------------------------------------
```

* Kết quả mô phỏng trên Active HDL 6.1



Hình 3.4 Kết quả mô phỏng cho ví dụ 3.3

Chương 4: Toán tử và thuộc tính.

4.1. Toán tử.

VHDL cung cấp một số toán tử sau:

- ✓ Toán tử gán.
- ✓ Toán tử logic.
- ✓ Toán tử toán học.
- ✓ Toán tử so sánh.
- ✓ Toán tử dịch.

Sau đây chúng ta sẽ xem xét cụ thể từng toán tử một.

4.1.1Toán tử gán.

VHDL định nghĩa ba loại toán tử gán sau:

<=: Dùng gán giá trị cho SIGNAL.

:= : Dùng gán giá trị cho VARIABLE, CONSTANT,GENERIC.

=>: Dùng gán giá trị cho thành phần các vector và các loại giá trị khác.

Ví dụ:

```
SIGNAL x : STD_LOGIC;
VARIABLE y : STD_LOGIC_VECTOR(3 DOWNTO 0);
SIGNAL w: STD_LOGIC_VECTOR(0 TO 7);
x <= '1';
y := "0000
w <= "10000000";
w <= (0 =>'1', OTHERS =>'0');
```

4.1.2Toán tử Logic.

VHDL định nghĩa các toán tử logic sau:

NOT, AND, OR, NAND, NOR, XOR, XNOR

Dữ liệu cho các toán tử này phải là kiểu: BIT, STD_LOGIC, STD_ULIGIC, BIT_VECTOR, STD_LOGIC_VECTOR, STD_ULOGIC_VECTOR.

Ví dụ:

```
y <= NOT a AND b;
y <= NOT (a AND b);
y <= a NAND b;
```

4.1.3Toán tử toán học.

Các toán tử này dùng cho các kiểu dữ liệu số như là:INTEGER, SIGNED, UNSIGNED, REAL. Các toán tử bao gồm:

\+ Toán tử cộng.

- Toán tử trừ.
* Toán tử nhân.
/ Toán tử chia.
** Toán tử lấy mũ.
MOD Phép chia lấy phần nguyên.
REM Phép chia lấy phần dư.
ABS Phép lấy giá trị tuyệt đối.

4.1.4Toán tử so sánh.

Có các toán tử so sánh sau:

= So sánh bằng
/= So sánh không bằng.
< So sánh nhỏ hơn.
> So sánh lớn hơn.
<= So sánh nhỏ hơn hoặc bằng.
>= So sánh lớn hơn hoặc bằng.

4.1.5Toán tử dịch.

Cú pháp sử dụng toán tử dịch là:

<left operand> <shift operation> <right operand>

Trong đó <left operand> có kiểu là BIT_VECTOR, còn <right operand> có kiểu là INTEGER. Có hai toán tử dịch:

Sll Toán tử dịch trái. Điền 0 vào phía phải.
Rll Toán tử dịch phải. Điền 0 vào phía trái.

4.2. Thuộc tính.

4.1.1.Thuộc tính dữ liệu.

VHDL cung cấp các thuộc tính sau.

d'LOW Trả về giá trị nhỏ nhất của chỉ số mảng.
d'HIGH Trả về chỉ số lớn nhất của mảng.
d'LEFT Trả về chỉ số bên trái nhất của mảng.
d'RIGHT Trả về chỉ số bên phải nhất của mảng.
d'LENGTH Trả về kích thước của mảng.
d'RANGE Trả về mảng chứa chỉ số.
d'REVERSE_RANGE Trả về mảng chứa chỉ số được đảo ngược.

Ví dụ: Nếu d là một vector được khai báo như sau:

```
SIGNAL d : STD_LOGIC_VECTOR(0 TO 7)
```

Ta sẽ có:

d'LOW = 0, d'HIGH = 7, d'LEFT = 7, d'RIGHT = 0, d'LENGTH = 8,
d'RANGE = (7 downto 0), d'REVERSE_RANGE = (0 to 7).

Các thuộc tính này có thể dùng trong các vòng lặp:

```
FOR i IN RANGE (0 TO 7) LOOP ...
FOR i IN x'RANGE LOOP ...
```

```
FOR i IN RANGE (x'LOW TO x'HIGH) LOOP ...
FOR i IN RANGE (0 TO x'LENGTH-1) LOOP ...
```

Nếu tín hiệu có kiểu liệt kê thì:

d'VAL(pos)	Trả về giá trị tại pos.
d'POS(val)	Trả về vị trí có giá trị là val.
d'LEFTOF(value)	Trả về giá trị ở vị trí bên trái của value.
d'VAL(row,colum)	Trả về giá trị ở một vị trí đặc biệt.

4.1.2.Thuộc tính tín hiệu.

Các thuộc tính loại này chỉ được áp dụng đối với dữ liệu SIGNAL. Nếu s là một SIGNAL thì ta có :

s'EVENT :	Trả về true khi một sự kiện xảy ra đối với s.
s'STABLE:	Trả về true nếu không có sự kiện nào xảy ra đối với s.
s'ACTIVE:	Trả về true khi s = 1.
s'QUIET<time>:	Trả về true khi trong khoảng thời gian time khong có sự kiện nào xảy ra.
s'LAST_EVENT:	Trả về thời gian trôi qua kể từ sự kiện cuối cùng
s'LAST_ACTIVE:	Trả về thới gian kể từ lần cuối cùng s = 1
s'LAST_VALUE:	Trả về giá trị của s trước sự kiện trước đó.

Trong các thuộc tính trên thì thuộc tính s'EVENT là hay được dùng nhất.

Vi dụ: Đây là ví dụ với tín hiệu đồng hồ.

```
IF (clk'EVENT AND clk='1')...
IF (NOT clk'STABLE AND clk='1')...
 WAIT UNTIL (clk'EVENT AND clk='1');
IF RISING_EDGE(clk)...
```

4.3. Thuộc tính được định nghĩa bởi người dùng.

VHDL, ngoài việc cung cấp các thuộc tính có sẵn nó còn cho phép người dùng tự định nghĩa các thuộc tính. Các thuộc tính này muốn sử dụng cần phải khai báo và mô tả rõ ràng theo cấu trúc sau:

```
ATTRIBUTE <attribute_name>:< attribute_type>;
ATTRIBUTE <attribute_name> OF< target_name>: <class>
IS    <value>;
```

Trong đó

+ attribute_type là kiểu dữ liệu.
+ Class : SIGNAL, TYPE, FUNCTION.

Ví dụ :

```
ATTRIBUTE number_of_inputs: INTEGER;
ATTRIBUTE number_of_inputs OF nand3: SIGNAL IS 3;
```

4.4. Chồng toán tử.

Cũng giống như các thuộc tính được định nghĩa bởi người dùng. Trong VHDL ta cũng có thể xây dựng chồng các toán tử toán học. Để xây dựng chồng các toán tử này ta cần phải chỉ rõ loại dữ liệu tham gia. Ví dụ như toán tử + ở trên chỉ áp dụng cho các loại dữ liệu cùng kiểu số.Bây giờ ta xây dựng toán tử + dùng để cộng một số INTEGER với một BIT.

```
FUNCTION "+" (a: INTEGER, b: BIT) RETURN INTEGER IS
BEGIN
      IF (b='1') THEN RETURN a+1;
      ELSE RETURN a;
      END IF;
END "+";
```

4.5. GENERIC.

GENERIC là một cách tạo các tham số dùng chung (giống như các biến static trong các ngôn ngữ lập trình). Mục đích là để cho các đoạn code mềm dẻo và dễ sử dụng lại hơn.

Một đoạn GENERIC khi được sử dụng cần phải được mô tả trong ENTITY. Các tham số phải được chỉ rõ. Cấu trúc như sau:

GENERIC (parameter_name : parameter_type := parameter_value);

Ví dụ: Ví dụ sau sẽ định nghĩa biến n có kiểu INTEGER và là GENERIC nó có giá trị mặc định là 8. Khi đó khi n được gọi ở bất kỳ đâu, trong một ENTITY hay một ARCHITECTURE theo sau đó giá trị của nó luôn là 8.

```
ENTITY my_entity IS
GENERIC (n : INTEGER := 8);
PORT (...);
END my_entity;
ARCHITECTURE my_architecture OF my_entity IS
...
END my_architecture;
```

Có thể có nhiều hơn 1 tham số GENERIC được mô tả trong một ENTITY. Ví dụ:

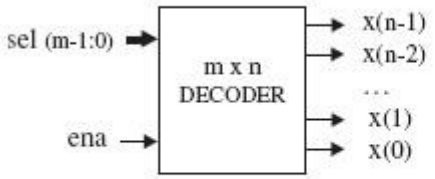
```
GENERIC (n: INTEGER := 8; vector: BIT_VECTOR := "00001111");
```

4.6. Ví dụ.

Để làm rõ hơn các vấn đề đã nói ở trên chúng ta sẽ xem xét một vài ví dụ sau:

Ví dụ 1: Generic Decoder.

Hình vẽ sau đây mô phỏng một bộ giải mã có hai đầu vào. Một tín hiệu vào dữ liệu sel gồm m bít và một tín hiệu là ena. Nó có một đầu ra dữ liệu gồm n bít. Có $m = \log_2(n)$.



ena	sel	x
0	00	1111
1	00	1110
	01	1101
	10	1011
	11	0111

DECODER

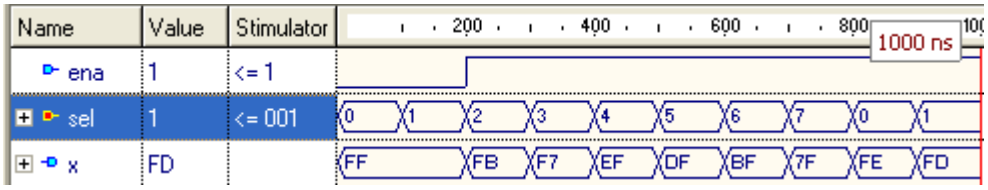
Hình 4.1. Bộ mã hoá cho ví dụ 4.1

Khi tín hiệu ena = '0' thì tất cả các bít đầu ra x = '1'. Đầu ra được chọn theo đầu vào sel. Chương trình sau đây mô tả về đối tượng này với 3 đầu vào sel và phía đầu ra có 8 đường x.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
ENTITY decoder IS
      PORT ( ena : IN STD_LOGIC;
            sel :  IN STD_LOGIC_VECTOR (2 DOWNTO
     0);
            x : OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END decoder;


ARCHITECTURE generic_decoder OF decoder IS
      BEGIN
            PROCESS (ena, sel)
            VARIABLE temp1 : STD_LOGIC_VECTOR
     (x'HIGH
                  DOWNTO 0);
            VARIABLE temp2 : INTEGER RANGE 0 TO
     x'HIGH;
      BEGIN
           temp1 := (OTHERS => '1');
            temp2 := 0;
            IF (ena='1') THEN
                 FOR i  IN sel'RANGE LOOP
                       IF (sel(i)='1') THEN
                            temp2:=2*temp2+1;
                        ELSE
                             temp2 := 2*temp2;
                        END IF;
                  END LOOP;
                  temp1(temp2):='0';
            END IF;
            x <= temp1;
            END PROCESS;
```

```
END generic_decoder;
```

Hình sau đây mô tả kết quả hoạt động của bộ giải mã trên.



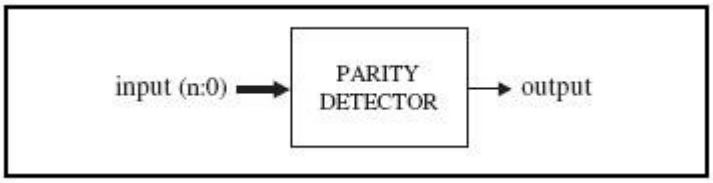
Hình 4.2 Mô phỏng kết quả của bộ mã hoá

Như chúng ta thấy khi ena =0 thì tất cả các bít phía đầu ra đều bằng 1. Khi ena = 1 thì chỉ một bít phía đầu ra được chọn tức là bằng 0. Ví dụ như khi sel=’000’ thì đầu ra x = ‘11111110’, sel = ‘001’ x = ‘11111101’….

Trong ví dụ trên ta đã sử dụng các toán tử +, * , các toán tử gán và thuộc tính RANGE.

Ví dụ 2 : Generic parity detector.

Ví dụ sau đây mô phỏng một mạch phát hiện tính parity. Nó bao gồm một đầu vào n bít và một đầu ra. Đầu ra sẽ có giá trị bằng 0 khi số đầu vào có giá trị là một là một số chẵn và bằng 1 trong các trường hợp còn lại.



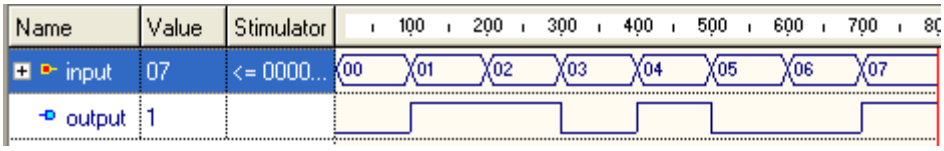
Hình 4.3. Bộ phát hiện bít chẵn lẻ

Sau đây là mã nguồn mô tả mạch trên.

```
ENTITY parity_det IS
      GENERIC (n : INTEGER := 7);
      PORT ( input: IN BIT_VECTOR (n DOWNTO 0);
                  output: OUT BIT);
 END parity_det;

 ARCHITECTURE parity OF parity_det IS
      BEGIN
            PROCESS (input)
                  VARIABLE temp: BIT;
                  BEGIN
                        temp := '0';
                        FOR i IN input'RANGE LOOP
```

```
                                    temp := temp XOR
                                input(i);
                                 END LOOP;
                                 output <= temp;
                        END PROCESS;
                END parity;
```

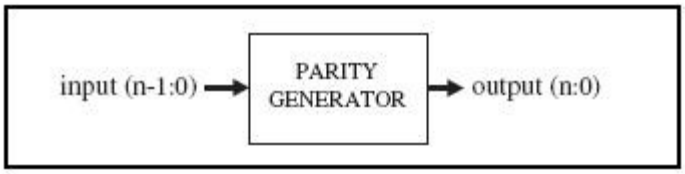
Trong đoạn mã trên chúng ta đã sử dụng một mệnh để GENERIC định nghĩa n =7. Khi đó tất cả các lần n xuất hiện nó đều có giá trị là 7.
Kết quả của mạch được biểu diễn bởi hình sau. Khi đầu vào input =’00000000’ thì đầu ra output =’0’. Khi input =’00000001’ thì đầu ra output = ‘1’ vì số đầu vào là 1 là một số lẻ.



Hình 4.4. Mô phỏng kết quả của hình 4.2

Ví dụ 3: Parity Generator

Mạch sau sẽ thêm một bít parity vào tín hiệu input. Bít này là 0 khi số đầu vào =1 của input là một số chẵn và bằng 0 trong trường hợp ngược lại. Như vậy mạch sẽ gồm n-1 đầu vào dữ liệu và n đầu ra, trong đó n-1 đầu ra bên phải giống như n-1 đầu vào, đầu ra còn lại là giá trị kiểm tra parity.

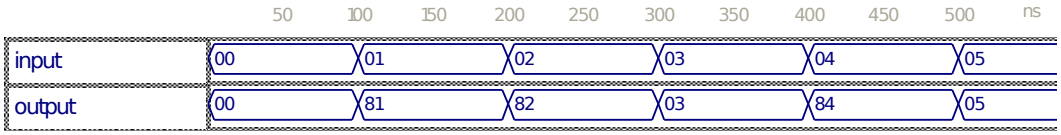


Hình 4.5. Bộ phát bit chẵn lẻ của ví dụ 4.3

```
ENTITY parity_gen IS
      GENERIC (n : INTEGER := 7);
      PORT ( input: IN BIT_VECTOR (n-1 DOWNTO 0);
           output: OUT BIT_VECTOR (n DOWNTO 0));
END parity_gen;


ARCHITECTURE parity OF parity_gen IS
      BEGIN
            PROCESS (input)
                  VARIABLE temp1: BIT;
                  VARIABLE temp2: BIT_VECTOR
            (output'RANGE);
            BEGIN
                  temp1 := '0';
                  FOR i IN input'RANGE LOOP
                        temp1 := temp1 XOR input(i);
                        temp2(i) := input(i);
```

```
                    END LOOP;
                    temp2(output'HIGH) := temp1;
                    output <= temp2;
                END PROCESS;
END parity;
```

Kết quả:



Hình 4.6. Mô phỏng kết quả của ví dụ 4.3

Như ta thấy khi đầu vào input =’0000000’ thì đầu ra output = ‘00000000’. Khi đầu vào input =’0000001’ thì đầu ra output = ‘10000001’.

Chương 5: Mã song song

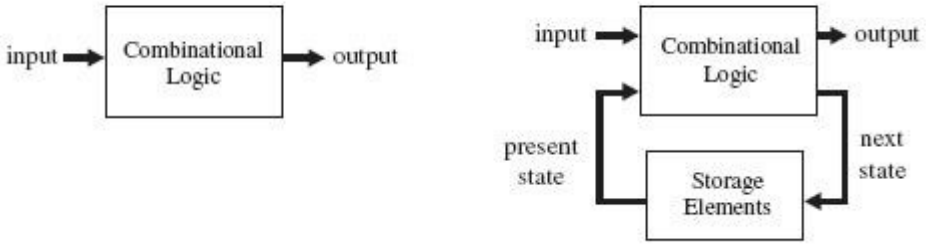
5.1. Song song và tuần tự.

Đầu tiên chúng ta sẽ xem xét sự khác biệt giữa mạch tổ hợp và mạch dãy sau đó sẽ xem sét sự khác biệt giữa mã nguồn tuần tự và mã song song.

5.1.1.Mạch tổ hợp và mạch dãy.

Mạch tổ hợp là mạch mà đầu ra của mạch chỉ phụ thuộc vào đầu vào của hệ tại thời điểm hiện tại. Từ đó ta thấy, hệ này không cần yêu câu bộ nhớ và chúng được tạo thành chỉ từ các cổng logic cơ bản.

Mạch dãy là mạch mà đầu ra của mạch còn phụ thuộc vào cả đầu vào trong quá khứ của mạch. Từ đó ta thấy đối với hệ này cần phải có bộ nhớ và một vòng phản hồi tín hiệu. Hình sau đây mô tả hai loại mạch này.



Hình 5.1. Mạch tổ hợp và mạch dãy

5.1.2.Mã song song và mã tuần tự.

Mã nguồn VHDL là song song. Chỉ các đoạn mã trong một PROCESS, FUNCTION, PROCEDURE là tuần tự. Các khối này được thực hiện một cách tuần tự. Mã song song đươc gọi là mã luồng dữ liệu (dataflow code).

Ví dụ. Một đoạn mã gồm ba khối lệnh song song (stat1, stat 2, stat3). Khi đó các đoạn sau sẽ thực hiện cùng một lúc trong mạch vật lý.

```
stat1      stat3      stat1
stat2  ≡   stat2  ≡   stat3  ≡  etc.
stat3      stat1      stat2
```

Các đoạn mã song song không thể sử dụng các thành phần của mạch đồng bộ (hoạt động chỉ xảy ra khi có sự đồng bộ của xung đồng hồ.). Một cách khác chúng ta chỉ có thể xây dựng dựa trên các mạch tổ hợp. Trong mục này chúng ta tìm hiểu về các đoạn mã song song. Chúng ta chỉ tìm hiểu các đoạn mã được sử dụng bên ngoài PROCESS, FUNCTION, PROCEDURES. Chúng là các khối lện WHEN và GENERATE. Bên cạnh đó, các phép gán dùng các toán tử được sử dụng để tạo các mạch tổ hợp. Cuối cùng một loại khối lện đặc biệt được gọi là BLOCK sẽ được sử dụng.

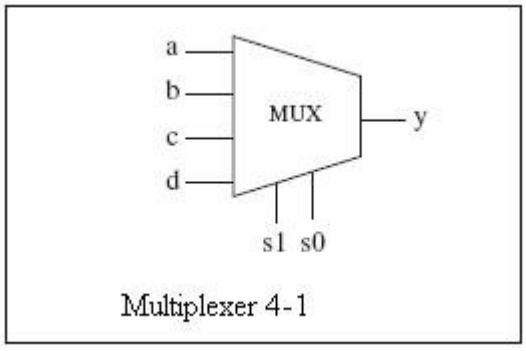
5.2. Sử dụng các toán tử.

Đây là cách cơ bản nhất dùng để tạo các đoạn mã song song. Các toán tử (AND, OR, ..) được tìm hiểu ở trên sẽ được liệt kê ở bảng dưới đây. Các toán tử có thể được sử dụng như là một thành phần của mạch tổ hợp. Tuy nhiên để rõ ràng. Các mạch hoàn chỉnh sẽ sử dụng cách viết tuần tự mặc dù các mạch không chứa các phần tử tuần tự. Các ví dụ sau đây được thiết kế chỉ sử dụng các thành phần logic cơ bản.

Operator type	Operators	Data types
Logical	NOT, AND, NAND, OR, NOR, XOR, XNOR	BIT, BIT_VECTOR, STD_LOGIC, STD_LOGIC_VECTOR, STD_ULOGIC, STD_ULOGIC_VECTOR
Arithmetic	+, –, *, /, ** (mod, rem, abs)	INTEGER, SIGNED, UNSIGNED
Comparison	=, /=, <, >, <=, >=	All above
Shift	sll, srl, sla, sra, rol, ror	BIT_VECTOR
Concatenation	&, (,,,)	Same as for logical operators, plus SIGNED and UNSIGNED

Bảng 5.1. Các toán tử

Ví dụ : Bộ dồn kênh 4 -1.

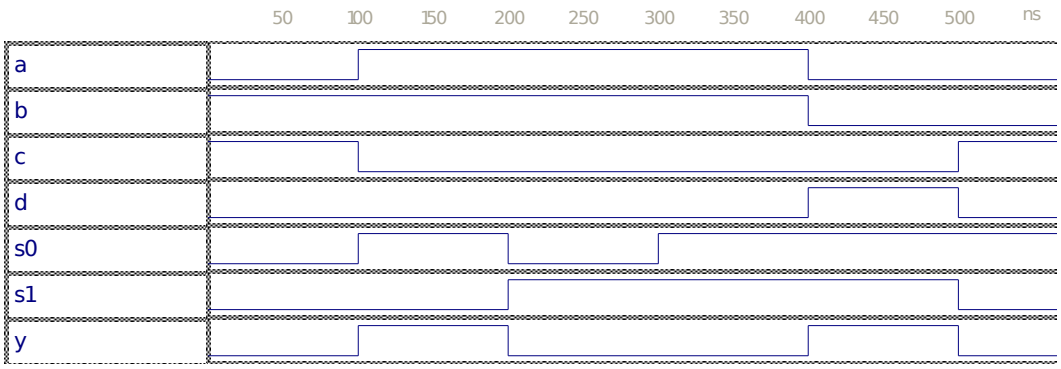


Hình 5.2. Bộ dồn kênh

Bộ dồn kênh 4-1 có 4 đầu vào dữ liệu, hai đầu vào điều khiển và một đầu ra. Tín hiệu đầu ra sẽ là tín hiệu của một trong 4 đầu vào tuỳ theo giá trị của hai đầu vào điều khiển s0,s1. Sau đây là chương trình mô phỏng.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
      PORT  (  a,  b,  c,  d,  s0,  s1:  IN
STD_LOGIC;
           y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE pure_logic OF mux IS
     BEGIN
          y <= (a AND NOT s1 AND NOT s0) OR
               (b AND NOT s1 AND s0) OR
                (c AND s1 AND NOT s0) OR
                 (d AND s1 AND s0);
END pure_logic;
```

Kết qủa mô phỏng.



Hình 5.3. Mô phỏng kết quả của ví dụ 5.1

5.3. Mệnh đề WHEN.

WHEN là môt thành phần của các khối lện song song. Nó xuất hiện trong hai trường hợp. WHEN / ELSE và WITH / SELECT / WHEN. Cú pháp được trình bấy như sau.

WHEN / ELSE:

```
assignment WHEN condition ELSE
assignment WHEN condition ELSE
...;
```

WITH / SELECT / WHEN:

```
WITH identifier SELECT
assignment WHEN value,
assignment WHEN value,
...;
```

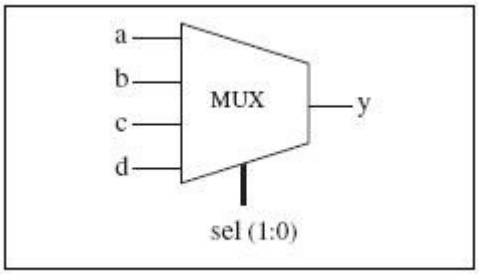
Ví dụ:

```
------              With                WHEN/ELSE
-------------------------
outp <= "000" WHEN (inp='0' OR reset='1') ELSE
"001" WHEN ctl='1' ELSE
"010";
----            With            WITH/SELECT/WHEN
--------------------
WITH control SELECT
output <= "000" WHEN reset,
"111" WHEN set,
UNAFFECTED WHEN OTHERS;
```

Sau đây ta sẽ xem xét các ví dụ dùng mệnh đề WHEN.

Ví dụ 1: Bộ dồn kênh 4 -1.

Nguyên tắc hoạt động của mạch này ta đã nói ở trên. Ở đây chúng ta sẽ dùng mệnh đề WHEN thay cho cá toán tử. Chúng ta có thể dùng theo cả hai cách. Để dễ hiểu chúng ta sẽ xem xét cả hai cách sử dụng mệnh đề WHEN.



Hình 5.4. Bộ dồn kệnh cho ví dụ 2

------- Sử dụng WHEN/ELSE --------

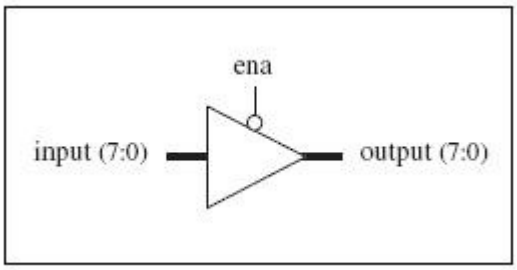
```
LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 ----------------------------------------
 ENTITY mux IS
       PORT ( a, b, c, d: IN STD_LOGIC;
              sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
              y: OUT STD_LOGIC);
 END mux;
 ----------------------------------------
 ARCHITECTURE mux1 OF mux IS
       BEGIN
              y <= a WHEN sel="00" ELSE
                   b WHEN sel="01" ELSE
                   c WHEN sel="10" ELSE
                   d;
 END mux1;
 ----------------------------------------
```

--- Sử dụng WITH/SELECT/WHEN -----

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY mux IS
        PORT ( a, b, c, d: IN STD_LOGIC;
                sel: IN STD_LOGIC_VECTOR (1 DOWNTO 0);
                y: OUT STD_LOGIC);
END mux;
-----------------------------------------
ARCHITECTURE mux2 OF mux IS
        BEGIN
                WITH sel SELECT
                        y <= a WHEN "00",
                        b WHEN "01",
                        c WHEN "10",
                        d WHEN OTHERS;
END mux2;
-----------------------------------------
```

Ví dụ 2: Bộ đệm 3 trạng thái.

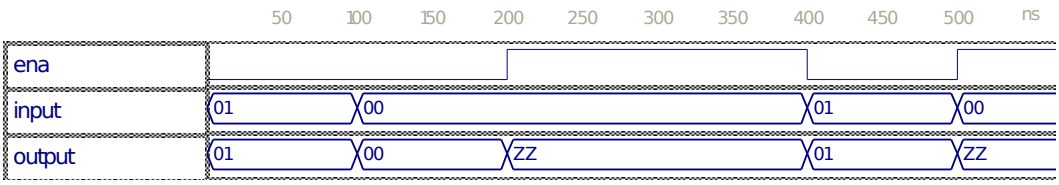


Hình 5.5. Bộ đệm 3 trạng thái

Mạch bộ đệm 3 trạng thái cho đầu ra output = input khi ena = 0 và trở kháng cao khi ena = 1.

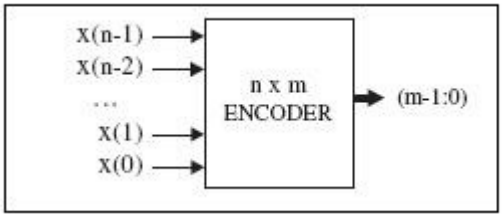
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY tri_state IS
      PORT ( ena: IN STD_LOGIC;
            input: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            output: OUT STD_LOGIC_VECTOR (7 DOWNTO
     0));
END tri_state;
-----------------------------------------------
ARCHITECTURE tri_state OF tri_state IS
      BEGIN
            output <= input WHEN (ena='0') ELSE
            (OTHERS => 'Z');
END tri_state;
-----------------------------------------------
```

Kết quả mô phỏng



Hình 5.6. Kết quả mô phỏng cho ví dụ 5.3

Ví dụ 3: Encoder.



Hình 5.7. Bộ mã hoá cho ví dụ 5.4

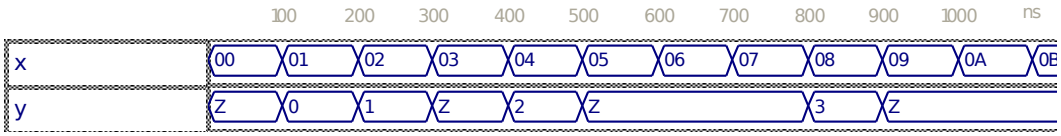
Một bộ ENCODER có n đầu vào, m đầu ra với m = $\log_2$ (n). Tại một thời điểm chỉ có một bít đầu vào bằng 1. Sau đây là chương trình mô phỏng sử dụng WHEN theo cả hai cách dùng WHEN / ELSE,và WITH / SELECT / WHEN.

---- sử dụng WHEN/ELSE ----------

```
LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 ---------------------------------------------
 ENTITY encoder IS
       PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
             y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
 END encoder;
 ---------------------------------------------
 ARCHITECTURE encoder1 OF encoder IS
       BEGIN
             y <=       "000" WHEN x="00000001" ELSE
                        "001" WHEN x="00000010" ELSE
                        "010" WHEN x="00000100" ELSE
                   "011" WHEN x="00001000" ELSE
                   "100" WHEN x="00010000" ELSE
                   "101" WHEN x="00100000" ELSE
                   "110" WHEN x="01000000" ELSE
                   "111" WHEN x="10000000" ELSE
                   "ZZZ";
 END encoder1;
 ---------------------------------------------
---- Sử dụng WITH/SELECT/WHEN ------
 LIBRARY ieee;
```

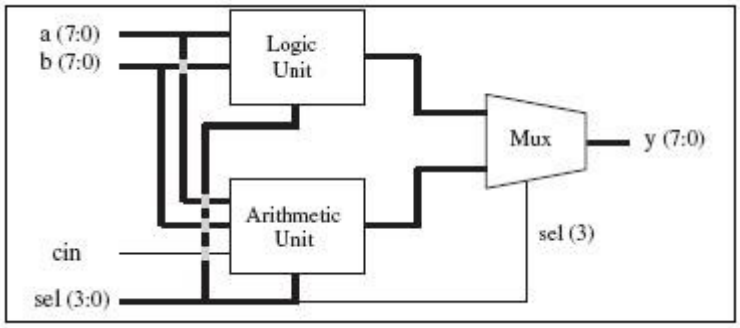
```
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY encoder IS
      PORT ( x: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
             y: OUT STD_LOGIC_VECTOR (2 DOWNTO 0));
END encoder;
-------------------------------------------
ARCHITECTURE encoder2 OF encoder IS
      BEGIN
            WITH x SELECT
                  y <= "000" WHEN "00000001",
                       "001" WHEN "00000010",
                       "010" WHEN "00000100",
                       "011" WHEN "00001000",
                       "100" WHEN "00010000",
                       "101" WHEN "00100000",
                       "110" WHEN "01000000",
                       "111" WHEN "10000000",
                       "ZZZ" WHEN OTHERS;
END encoder2;
-------------------------------------------
```

Kết quả mô phỏng:



Hình 5.8. Kết quả mô phỏng cho ví dụ 5.4

Ví dụ 4: ALU.



Hình 5.9. ALU

Mạch ALU thực hiện các phép toán logic và toán học đối với hai đầu vào a và b. Chúng được điều khiển bởi 4 bít sel(3:0). Tuỳ thuộc vào giá trị của sel mà khối sẽ thực hiện thao tác nào với dữ liệu. Bảng dưới đây mô tả các thao tác của ALU.

sel	Operation	Function	Unit
0000	y <= a	Transfer a	Arithmetic
0001	y <= a+1	Increment a	
0010	y <= a-1	Decrement a	
0011	y <= b	Transfer b	
0100	y <= b+1	Increment b	
0101	y <= b-1	Decrement b	
0110	y <= a+b	Add a and b	
0111	y <= a+b+cin	Add a and b with carry	
1000	y <= NOT a	Complement a	Logic
1001	y <= NOT b	Complement b	
1010	y <= a AND b	AND	
1011	y <= a OR b	OR	
1100	y <= a NAND b	NAND	
1101	y <= a NOR b	NOR	
1110	y <= a XOR b	XOR	
1111	y <= a XNOR b	XNOR	

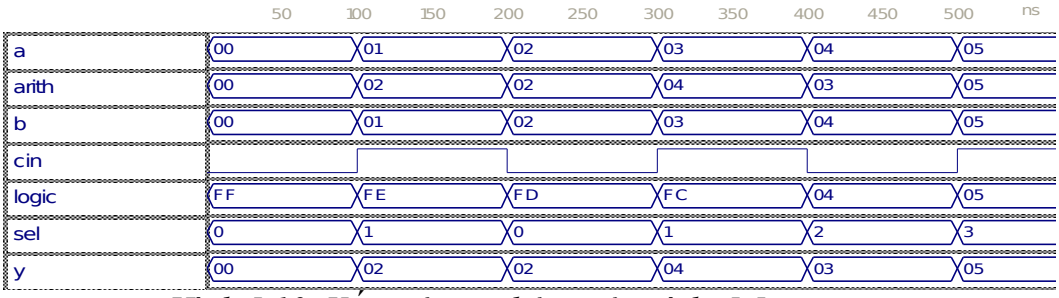
Hình 5.9.b. Hoạt động chinh của các phần tử ALU

Mã nguồn thực hiện mô phỏng:

```
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_unsigned.all;
-------------------------------------------
ENTITY ALU IS
      PORT (a, b: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            sel: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
            cin: IN STD_LOGIC;
            y: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END ALU;
-------------------------------------------
ARCHITECTURE dataflow OF ALU IS
       SIGNAL  arith,  logic:  STD_LOGIC_VECTOR  (7
DOWNTO 0);
BEGIN
      ----- Arithmetic unit: ------
      WITH sel(2 DOWNTO 0) SELECT
      arith <= a WHEN "000",
            a+1 WHEN "001",
            a-1 WHEN "010",
            b WHEN "011",
            b+1 WHEN "100",
            b-1 WHEN "101",
            a+b WHEN "110",
            a+b+cin WHEN OTHERS;
      ----- Logic unit: -----------
      WITH sel(2 DOWNTO 0) SELECT
            logic <= NOT a WHEN "000",
                  NOT b WHEN "001",
                  a AND b WHEN "010",
                  a OR b WHEN "011",
                  a NAND b WHEN "100",
                  a NOR b WHEN "101",
```

```
                    a XOR b WHEN "110",
                    NOT (a XOR b) WHEN OTHERS;
        -------- Mux: ---------------
        WITH sel(3) SELECT
              y <= arith WHEN '0',
              logic WHEN OTHERS;
END dataflow;
-----------------------------------------------
```

Kết quả mô phỏng.



Hình 5.10. Kết quả mô phỏng của ví dụ 5.5

5.4. GENERATE.

GENERATE là một khối lệnh song song khác. Nó tương đương với khối lệnh tuần tự LOOP trong việc cho phép các đoạn lệnh được thực hiện lặp lại một số lần nào đó. Mẫu dùng của nó là FOR / GENERATE.

```
label: FOR identifier IN range GENERATE
(concurrent assignments)
END GENERATE;
```

Một cách khác sử dụng GENERATE là dùng IF. Ở đây mệnh đề ELSE không được sử dụng. Một cách hay được sử dụng là dùng IF trong FOR/GENERATE.

Mẫu sử dụng như sau.

```
label1: FOR identifier IN range GENERATE
...
label2: IF condition GENERATE
(concurrent assignments)
END GENERATE;
...
END GENERATE;
```

Ví dụ:

```
SIGNAL x: BIT_VECTOR (7 DOWNTO 0);
SIGNAL y: BIT_VECTOR (15 DOWNTO 0);
SIGNAL z: BIT_VECTOR (7 DOWNTO 0);
...
G1: FOR i IN x'RANGE GENERATE
z(i) <= x(i) AND y(i+8);
END GENERATE;
```

Một điều cần phải chú ý là giới hạn của dãy phải được khai báo là static nếu không sẽ không hợp lệ. Trong ví dụ sau choice không được khai báo là static nên không hợp lệ:

```
NotOK: FOR i IN 0 TO choice GENERATE
(concurrent statements)
```

```
END GENERATE;
```

Để hiểu rõ hơn về khối lệnh GENERATE chúng ta sẽ xét ví dụ sau:

Ví dụ: Vector shifter.

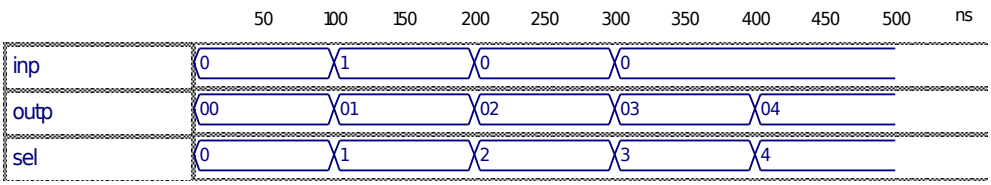
Ví dụ sau minh hoạ cho việc sử dụng GENERATE. Trong đó đầu vào sẽ được dịch đi một bít và tạo thành đầu ra. Ví dụ đầu vào có 4 đường và giá trị ban đầu là 1111 thì đầu ra sẽ được mô tả như sau:

row(0): 0 0 0 0 1 1 1 1
row(1): 0 0 0 1 1 1 1 0
row(2): 0 0 1 1 1 1 0 0
row(3): 0 1 1 1 1 0 0 0
row(4): 1 1 1 1 0 0 0 0

Chương trình mô phỏng.

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY shifter IS
PORT ( inp: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
       sel: IN INTEGER RANGE 0 TO 4;
      outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END shifter;
-------------------------------------------------
ARCHITECTURE shifter OF shifter IS
SUBTYPE vector IS STD_LOGIC_VECTOR (7 DOWNTO 0);
TYPE matrix IS ARRAY (4 DOWNTO 0) OF vector;
SIGNAL row: matrix;
BEGIN
       row(0) <= "0000" & inp;
      G1: FOR i IN 1 TO 4 GENERATE
       row(i) <= row(i-1)(6 DOWNTO 0) & '0';
END GENERATE;
outp <= row(sel);
END shifter;
```

Kết quả mô phỏng:



Hình 5.11. Kết quả mô phỏng của ví dụ 5.6

Như hình ta thấy, nếu input = “0011”thì đầu ra output =”00000011” khi sel = 0. output = “00000110” khi sel = 1. output = 00001100 nếu sel = 2.

5.5. BLOCK.

Có hai loại khối lệnh BLOCK : Simple và Guarded.

5.5.1.Simple BLOCK

Khối lệnh BLOCK cho phép đặt một khối lệnh song song vào một đoạn, điều đó giúp cho các đoạn lệnh dễ đọc và dễ quản lý hơn. Cấu trúc của chúng như sau:

```
label: BLOCK
      [declarative part]
      BEGIN
            (concurrent statements)
END BLOCK label;
```

Các khối lệnh BLOCK đặt liên tiếp nhau như ví dụ sau:

```
----------------------
ARCHITECTURE example ...
BEGIN
...
block1: BLOCK
BEGIN
...
END BLOCK block1
...
block2: BLOCK
BEGIN
...
END BLOCK block2;
...
END example;
------------------------
```

Ví dụ:

```
b1: BLOCK
SIGNAL a: STD_
BEGIN
a <= input_sig
END BLOCK b1;
```

Một đoạn BLOCK có thể được đặt trong một đoạn BLOCK khác, khi đó cú pháp như sau:

```
label1: BLOCK
      [declarative part of top block]
BEGIN
      [concurrent statements of top block]
label2: BLOCK
      [declarative part nested block]
BEGIN
      (concurrent statements of nested block)
END BLOCK label2;
      [more concurrent statements of top block]
END BLOCK label1;
```

5.5.2.Guarded BLOCK

Một Guarded BLOCK là một khối BLOCK đặc biệt. Nó chứa một điều kiện và BLOCK chỉ được thực hiện khi điều kiện đó có giá trị là TRUE.
Cấu trúc như sau:

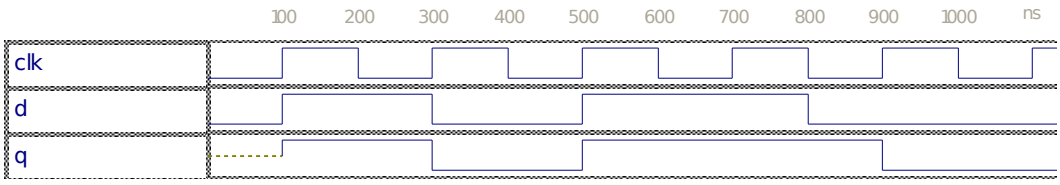
```
label: BLOCK (guard expression)
     [declarative part]
BEGIN
     (concurrent guarded and unguarded
statements)
END BLOCK label;
```

Để tìm hiểu rõ hơn về khối BLOCK ta đi xét ví dụ sau:

Vídụ 1: Chốt sử dụng Guarded BLOCK. Trong ví dụ này khi nào clk = '1' thì khối đươc hoạt động khi đó khối lệnh sẽ đươc thực hiện.

```
-------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY latch IS
      PORT (d, clk: IN STD_LOGIC;
            q: OUT STD_LOGIC);
END latch;
-------------------------------
ARCHITECTURE latch OF latch IS
      BEGIN
            b1: BLOCK (clk='1')
            BEGIN
                  q <= GUARDED d;
            END BLOCK b1;
      END latch;
-------------------------------
```

Kết quả mô phỏng



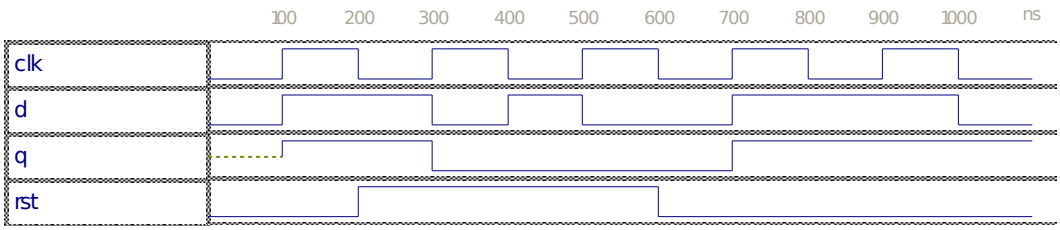
Hình 5.12. Kết quả mô phỏng cho ví dụ 5.7

Ví dụ 2: DFF dùng Guarded BLOCK.Trong ví dụ này chúng ta sẽ xem xét hoạt động của một TrigerT hoạt động đồng bộ sườn dương.

```
-------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------
ENTITY dff IS
      PORT ( d, clk, rst: IN STD_LOGIC;
            q: OUT STD_LOGIC);
END dff;
-------------------------------
ARCHITECTURE dff OF dff IS
```

```
                BEGIN
                      b1: BLOCK (clk'EVENT AND clk='1')
                      BEGIN
                            q <= GUARDED '0' WHEN rst='1'
                          ELSE d;
                      END BLOCK b1;
          END dff;
          ------------------------------
```

Kết quả mô phỏng:



Hình 5.13. Kết quả mô phỏng của ví dụ 5.8

Chương 6: Mã tuần tự

6.1. PROCESS

PROCESS là phần tuần tự của mã VHDL. Nó được mô tả bởi các câu lệnh IF, WAIT, CASE, hoặc LOOP, và bởi danh sách nhạy (ngoại trừ WAIT được sử dụng). PROCESS phải được cài đặt trong mã chính, và được thực thi ở mọi thời điểm một tín hiệu trong danh sách nhạy thay đổi.

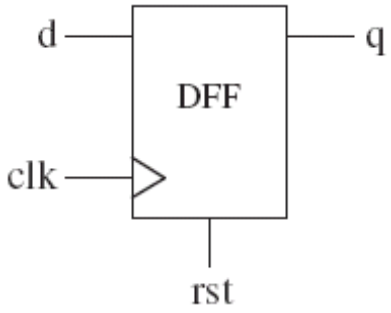
Cú pháp:

```
[label:]  PROCESS (sensitivity list)
               [VARIABLE name type [range] [:=
          initial_value;]]
          BEGIN
               (sequential code)
          END PROCESS [label];
```

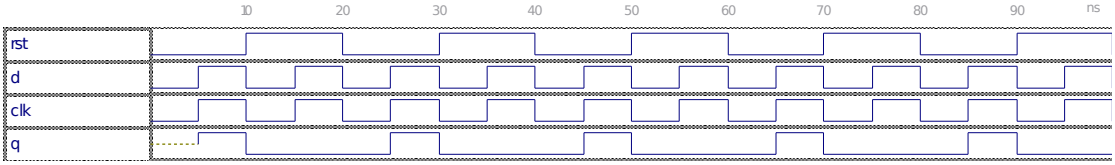
VARIABLES là tuỳ chọn. Nếu sử dụng, chúng phải được khai báo trong phần khai báo của PROCESS (trước từ khoá BEGIN). Giá trị khởi tạo không thể kết hợp, chỉ lấy để đại diện khi mô phỏng.

Nhãn cũng được sử dụng tuỳ chọn, mục đích là nâng cao khả năng đọc được của mã. Nhãn có thể là bất kỳ từ nào, ngoại trừ từ khoá.

Ví dụ 6.1a:



Hình 6.1a.1 DFF với tín hiệu reset không đồng bộ



Hình 6.1a.2 Kết quả mô phỏng

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DFF is
      Port(d,clk,rst:in std_logic;
            q:out std_logic);
end DFF;

architecture Behaviour of DFF is
begin
      process(clk,rst)
      begin
            -- wait on rst,clk;
            if (rst='1') then
                  q <= '0';
            elsif (clk'Event and clk='1') then
                  q <= d;
            end if;
      end process;
end Behaviour;
```

6.2. Signals và Variables.

VHDL có hai cách định nghĩa các giá trị không tĩnh: bằng SIGNAL hoặc bằng VARIABLE. SIGNAL có thể được khai báo trong PACKAGE, ENTITY hoặc ARCHITECTURE (trong phần khai báo của nó), trong khi VARIABLE có thể được mô tả bên trong một phần của mã tuần tự (trong PROCESS). Do đó, trong khi giá trị của phần ở trước có thể là toàn cục, phần ở sau luôn là cục bộ.

Giá trị của VARIABLE có thể không bao giờ định nghĩa ngoài PROCESS một cách trực tiếp, nếu cần, thì nó phải được gán thành SIGNAL.

Trong cách xử lý khác, cập nhật VARIABLE là tức thì, ta có thể tính toán tức thì giá trị mới của nó trong dòng lệnh tiếp theo. Điều đó không phải là trường hợp của SIGNAL (khi được sử dụng trong PROCESS), giá trị mới của nó chỉ tổng quát được bảo toàn để có thể dùng được sau khi kết thúc quá trình chạy hiện tại của PROCESS.

Phép toán gán cho SIGNAL là "<=" (sig <= 5), trong khi với VARIABLE là ":=" (var := 5).

6.3. IF.

IF, WAIT, CASE, và LOOP là các câu lệnh đối với mã tuần tự. Do đó, chúng chỉ có thể được sử dụng bên trong PROCESS, FUNCTION hoặc PROCEDURE.

Về nguyên tắc, có một kết quả phủ định, tổng hợp sẽ tối ưu hoá cấu trúc và tránh đi sâu vào phần cứng.

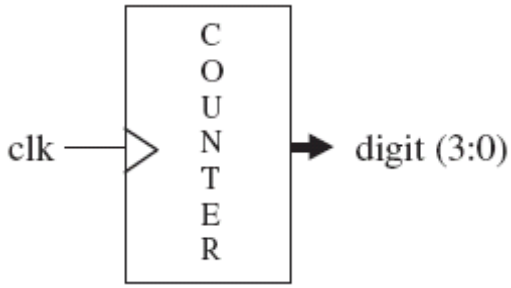
Cú pháp:

```
IF conditions THEN assignments;
ELSIF conditions THEN assignments;
...
ELSE assignments;
END IF;
```

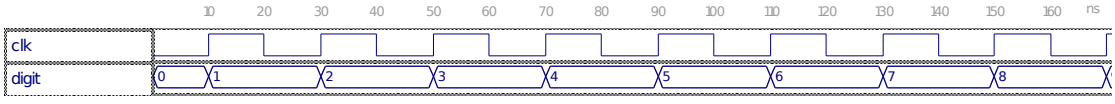
Ví dụ:

```
IF (x<y) THEN temp:="11111111";
ELSIF (x=y AND w='0') THEN temp:="11110000";
ELSE temp:=(OTHERS =>'0');
```

Ví dụ 6.3a:



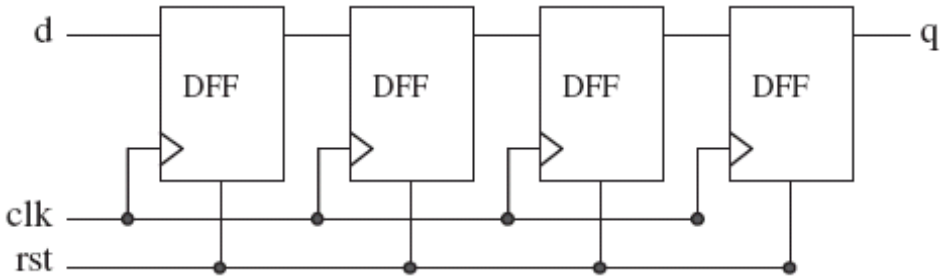
Hình 6.2a.1. Bộ đếm chữ số thập phân



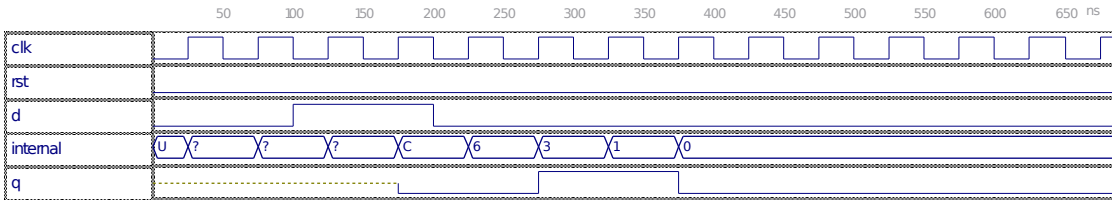
Hình 6.2a.2. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counter IS
      PORT (clk : IN STD_LOGIC;
                  digit : OUT INTEGER RANGE 0 TO 9);
END counter;

ARCHITECTURE counter OF counter IS
BEGIN
      count: PROCESS(clk)
      VARIABLE temp : INTEGER RANGE 0 TO 10;
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  temp := temp + 1;
                  IF (temp=10) THEN temp := 0;
                  END IF;
            END IF;
            digit <= temp;
      END PROCESS count;
END counter;
```

Ví dụ 6.3b:



Hình 6.3b.1. Thanh ghi dịch 4 bit



Hình 6.3b.2. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shiftreg IS
```

```
      GENERIC (n: INTEGER := 4); -- # of stages
      PORT (d, clk, rst: IN STD_LOGIC;
                  q: OUT STD_LOGIC);
END shiftreg;

ARCHITECTURE behavior OF shiftreg IS
      SIGNAL internal: STD_LOGIC_VECTOR (n-1 DOWNTO 0);
BEGIN
      PROCESS (clk, rst)
      BEGIN
            IF (rst='1') THEN
                  internal <= (OTHERS => '0');
            ELSIF (clk'EVENT AND clk='1') THEN
                  internal <= d & internal(internal'LEFT DOWNTO
1);
            END IF;
      END PROCESS;
      q <= internal(0);
END behavior;
```

6.4. WAIT.

Phép toán WAIT đôi khi tương tự như IF. Tuy nhiên, nhiều hơn một định dạng có thể dùng được. Hơn nữa, khi IF, CASE, hoặc LOOP được sử dụng, PROCESS không thể có một danh sách nhạy khi WAIT được sử dụng.

Cú pháp:

```
WAIT UNTIL signal_condition;

WAIT ON signal1 [, signal2, ... ];

WAIT FOR time;
```

Câu lệnh WAIT UNTIL nhận chỉ một tín hiệu, do đó thích hợp cho mã đồng bộ hơn là mã không đồng bộ. Khi PROCESS không có danh sách nhạy trong trường hợp này, WAIT phải là câu lệnh đầu tiên trong PROCESS. PROCESS được thực hiện mọi thời điểm khi gặp điều kiện.

Ví dụ:

Thanh ghi 8 bit với tín hiệu reset đồng bộ

```
PROCESS -- no sensitivity list
BEGIN
      WAIT UNTIL (clk'EVENT AND clk='1');
      IF (rst='1') THEN
            output <= "00000000";
      ELSIF (clk'EVENT AND clk='1') THEN
            output <= input;
      END IF;
```

```
END PROCESS;
```

WAIT ON, trong cách xử lý khác, nhận nhiều tín hiệu. PROCESS được đặt giữ cho đến khi bất kỳ tín hiệu nào thay đổi. PROCESS sẽ tiếp tục thực hiện bất kỳ khi nào một thay đổi trong rst hoặc clk xuất hiện.

Ví dụ:

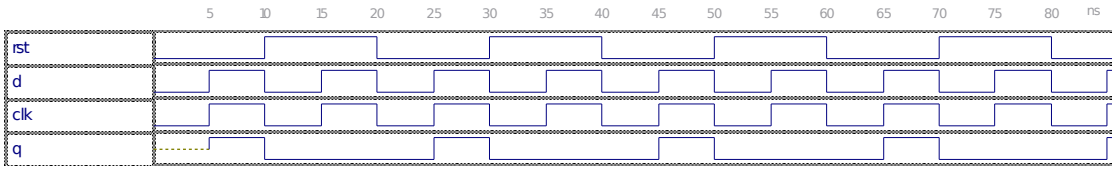
Thanh ghi 8 bit với tín hiệu reset không đồng bộ

```
PROCESS
BEGIN
      WAIT ON clk, rst;
      IF (rst='1') THEN
            output <= "00000000";
      ELSIF (clk'EVENT AND clk='1') THEN
            output <= input;
      END IF;
END PROCESS;
```

WAIT FOR chỉ dùng để mô phỏng. Ví dụ: WAIT FOR 5ns;

Ví dụ 6.4a:

DFF với tín hiệu reset không đồng bộ



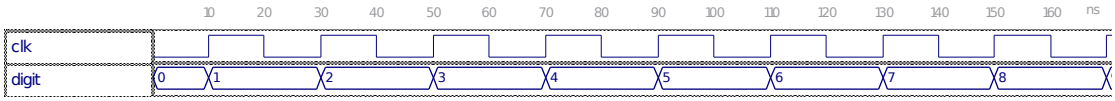
Hình 6.4a.1. Kết quả mô phỏng

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;

entity DFF is
      Port(d,clk,rst:in std_logic;
            q:out std_logic);
end DFF;

architecture DFF of DFF is
begin
      process
      begin
            wait on rst,clk;
```

```
            if (rst='1') then
                  q <= '0';
            elsif (clk'Event and clk='1') then
                  q <= d;
            end if;
      end process;
end DFF;
```

Ví dụ 6.4b:

Bộ đếm một chữ số thập phân $0 \rightarrow 9 \rightarrow 0$



Hình 6.4b.1. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counter IS
      PORT (clk : IN STD_LOGIC;
                  digit : OUT INTEGER RANGE 0 TO 9);
END counter;

ARCHITECTURE counter OF counter IS
BEGIN
      PROCESS -- no sensitivity list
            VARIABLE temp : INTEGER RANGE 0 TO 10;
      BEGIN
            WAIT UNTIL (clk'EVENT AND clk='1');
            temp := temp + 1;
            IF (temp=10) THEN
                  temp := 0;
            END IF;
            digit <= temp;
      END PROCESS;
END counter;
```

6.5. CASE.

CASE là lệnh duy nhất cho mã tuần tự (đi kèm với IF, LOOP, và WAIT).

Cú pháp:

```
CASE identifier IS
WHEN value => assignments;
WHEN value => assignments;
...
END CASE;
```

Ví dụ:

```
CASE control IS
      WHEN "00" => x<=a; y<=b;
      WHEN "01" => x<=b; y<=c;
      WHEN OTHERS => x<="0000"; y<="ZZZZ";
END CASE;
```

Lệnh CASE (tuần tự) tương tự với WHEN (kết hợp). Tất cả sự hoán vị đều phải được kiểm tra, vì vậy từ khoá OTHERS rất hữu ích. Từ khoá quan trọng khác là NULL (bản sao của UNAFFECTED), nên được sử dụng khi không có hoạt động nào thay thế. Ví dụ: WHEN OTHERS => NULL;. Tuy nhiên, CASE cho phép nhiều phép gán với mỗi điều kiện kiểm tra, trong khi WHEN chỉ cho phép một.

Giống như trong trường hợp của WHEN, ở đây "WHEN value" có thể có 3 dạng:

```
WHEN value -- single value
WHEN value1 to value2 -- range, for enumerated data types
                      -- only
WHEN value1 | value2 |... -- value1 or value2 or ...
```

	WHEN	CASE
Kiểu lệnh	Đồng thời	Tuần tự
Sử dụng	Chỉ ngoài PROCESS, FUNCTION, hoặc PROCEDURE	Chỉ trong PROCESS, FUNCTION, hoặc PROCEDURE
Tất cả sự hoán vị phải được kiểm tra	Có với WITH/SELECT/WHEN	Có
Số phép gán lớn nhất cho mỗi kiểm tra	1	Bất kỳ
Từ khoá không kích hoạt	UNAFFECTED	NULL

Bảng 6.1. So sánh giữa WHEN và CASE

Ví dụ:

Với WHEN:

```
WITH sel SELECT
      x <= a WHEN "000",
      b WHEN "001",
      c WHEN "010",
UNAFFECTED WHEN OTHERS;
```

Với CASE:

```
CASE sel IS
```

```
      WHEN "000" => x<=a;
      WHEN "001" => x<=b;
      WHEN "010" => x<=c;
      WHEN OTHERS => NULL;
END CASE;
```

Ví dụ:

Bộ dồn kênh MUX 4-1

Với IF:

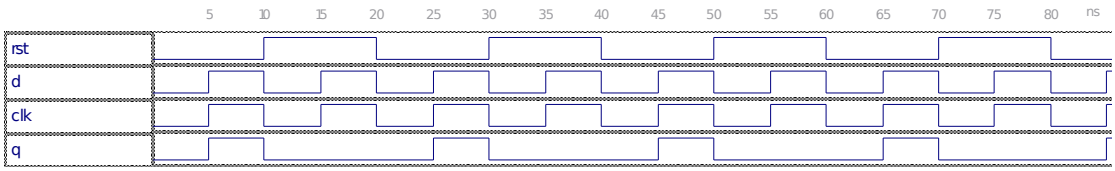
```
IF (sel="00") THEN x<=a;
ELSIF (sel="01") THEN x<=b;
ELSIF (sel="10") THEN x<=c;
ELSE x<=d;
```

Với CASE:

```
CASE sel IS
      WHEN "00" => x<=a;
      WHEN "01" => x<=b;
      WHEN "10" => x<=c;
      WHEN OTHERS => x<=d;
END CASE;
```

Ví dụ 6.5a:

DFF với tín hiệu reset không đồng bộ



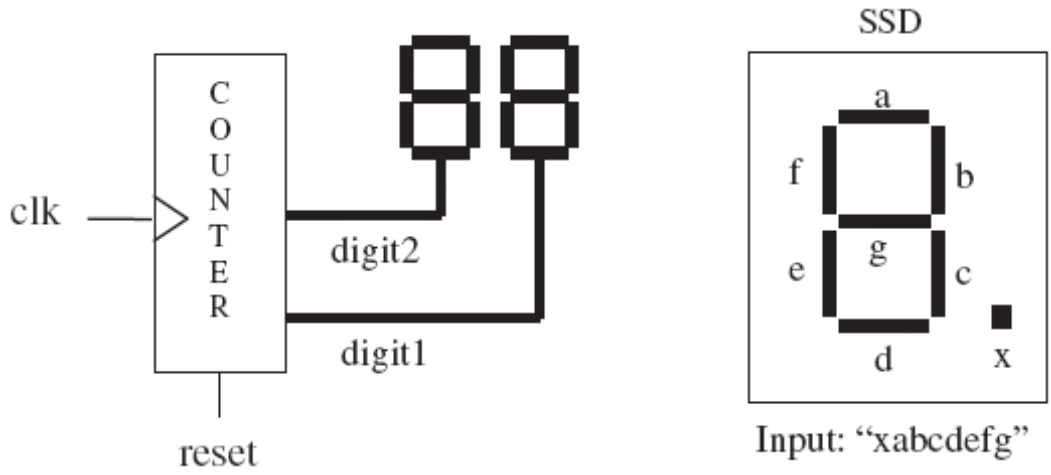
Hình 6.5a.1. Kết quả mô phỏng

```
LIBRARY ieee; -- Unnecessary declaration, -- because
USE ieee.std_logic_1164.all; -- BIT was used instead of
                             -- STD_LOGIC
ENTITY dff IS
      PORT (d, clk, rst: IN BIT;
                  q: OUT BIT);
END dff;
ARCHITECTURE dff3 OF dff IS
BEGIN
      PROCESS (clk, rst)
      BEGIN
```

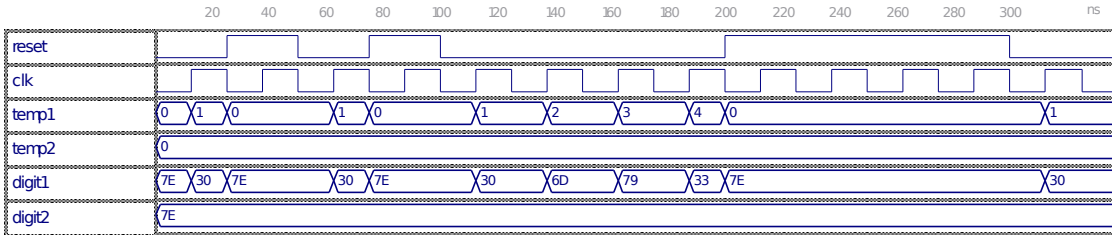
```
            CASE rst IS
                  WHEN '1' => q<='0';
                  WHEN '0' =>
                        IF (clk'EVENT AND clk='1') THEN
                              q <= d;
                        END IF;
                  WHEN OTHERS => NULL;-- Unnecessary,rst is of
                                       -- type BIT
            END CASE;
      END PROCESS;
END dff3;
```

Ví dụ 6.5b:

Bộ đếm hai chữ số thập phân 0 → 99 → 0, đầu ra là 2 LED 7 thanh



Hình 6.5b.1. Bộ đếm 2 chữ số thập phân



Hình 6.5b.2. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY counter IS
      PORT (clk, reset : IN STD_LOGIC;
            digit1, digit2 : OUT STD_LOGIC_VECTOR (6 DOWNTO 0));
END counter;

ARCHITECTURE counter OF counter IS
BEGIN
```

```
      PROCESS(clk, reset)
            VARIABLE temp1: INTEGER RANGE 0 TO 10;
            VARIABLE temp2: INTEGER RANGE 0 TO 10;
      BEGIN
      ---------------------- counter: ----------------------
            IF (reset='1') THEN
                  temp1 := 0;
                  temp2 := 0;
            ELSIF (clk'EVENT AND clk='1') THEN
                  temp1 := temp1 + 1;
                  IF (temp1=10) THEN
                        temp1 := 0;
                        temp2 := temp2 + 1;
                        IF (temp2=10) THEN
                              temp2 := 0;
                        END IF;
                  END IF;
            END IF;
            ---- BCD to SSD conversion: --------
            CASE temp1 IS
                  WHEN 0 => digit1 <= "1111110"; --7E
                  WHEN 1 => digit1 <= "0110000"; --30
                  WHEN 2 => digit1 <= "1101101"; --6D
                  WHEN 3 => digit1 <= "1111001"; --79
                  WHEN 4 => digit1 <= "0110011"; --33
                  WHEN 5 => digit1 <= "1011011"; --5B
                  WHEN 6 => digit1 <= "1011111"; --5F
                  WHEN 7 => digit1 <= "1110000"; --70
                  WHEN 8 => digit1 <= "1111111"; --7F
                  WHEN 9 => digit1 <= "1111011"; --7B
                  WHEN OTHERS => NULL;
            END CASE;
            CASE temp2 IS
                  WHEN 0 => digit2 <= "1111110"; --7E
                  WHEN 1 => digit2 <= "0110000"; --30
                  WHEN 2 => digit2 <= "1101101"; --6D
                  WHEN 3 => digit2 <= "1111001"; --79
                  WHEN 4 => digit2 <= "0110011"; --33
                  WHEN 5 => digit2 <= "1011011"; --5B
                  WHEN 6 => digit2 <= "1011111"; --5F
                  WHEN 7 => digit2 <= "1110000"; --70
                  WHEN 8 => digit2 <= "1111111"; --7F
                  WHEN 9 => digit2 <= "1111011"; --7B
                  WHEN OTHERS => NULL;
            END CASE;
      END PROCESS;
END counter;
```

6.6. LOOP.

LOOP hữu ích khi một phần của mã phải được thể hiện nhiều lần. Giống như IF, WAIT, và CASE, LOOP là duy nhất đối với mã tuần tự, vì vậy nó cũng có thể được sử dụng bên trong PROCESS, FUNCTION, hay PROCEDURE.

Có nhiều cách sử dụng LOOP.

Cú pháp:

FOR/LOOP: vòng lặp được lặp lại một số lần cố định.

```
[label:]   FOR identifier IN range LOOP
                (sequential statements)
           END LOOP [label];
```

WHILE/LOOP: vòng lặp được lặp cho đến khi điều kiện không thảo mãn.

```
[label:]   WHILE condition LOOP
                (sequential statements)
           END LOOP [label];
```

EXIT: sử dụng để kết thúc vòng lặp.

```
[label:]   EXIT [label] [WHEN condition];
```

NEXT: sử dụng để bỏ qua các bước vòng lặp.

```
[label:]   NEXT [loop_label] [WHEN condition];
```

Ví dụ:

Với FOR/LOOP:

```
FOR i IN 0 TO 5 LOOP
     x(i) <= enable AND w(i+2);
     y(0, i) <= w(i);
     END LOOP;
```

Một đặc điểm quan trọng của FOR/LOOP (tương tự tạo với GENERATE) là giới hạn tối thiểu phải là tĩnh. Do đó, một khai báo kiểu "FOR i IN 0 TO choice LOOP", với choice là một tham số đầu vào (không tĩnh), không kết hợp tổng quát được.

Ví dụ:

Với WHILE/LOOP

```
WHILE (i < 10) LOOP
     WAIT UNTIL clk'EVENT AND clk='1';
     (other statements)
END LOOP;
```

Ví dụ:

Với EXIT

```
FOR i IN data'RANGE LOOP
     CASE data(i) IS
          WHEN '0' => count:=count+1;
```

```
            WHEN OTHERS => EXIT;
      END CASE;
END LOOP;
```

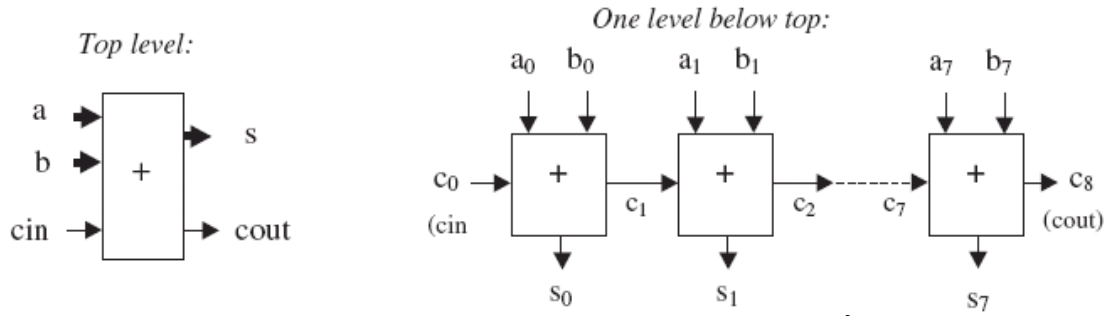
Ví dụ:

Với NEXT

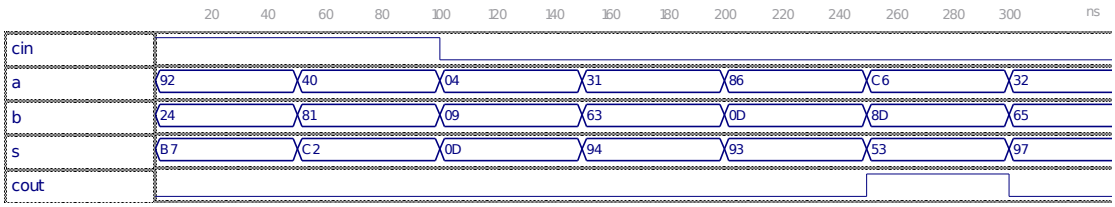
```
FOR i IN 0 TO 15 LOOP
      NEXT WHEN i=skip; -- jumps to next iteration
      (...)
END LOOP;
```

Ví dụ 6.6a:

Bộ cộng có nhớ 8 bit không dấu.



Hình 6.6a.1. Bộ cộng có nhớ 8 bit không dấu



Hình 6.6a.2. Kết quả mô phỏng

Mỗi phần tử của sơ đồ là một bộ cộng đầy đủ.

```
sj = aj XOR bj XOR cj
cjþ1 = (aj AND bj) OR (aj AND cj) OR (bj AND cj)
```

Cách 1:

Dùng Generic với các VECTOR

----- Solution 1: Generic, with VECTORS --------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder IS
      GENERIC (length : INTEGER := 8);
      PORT ( a, b: IN STD_LOGIC_VECTOR (length-1 DOWNTO 0);
                  cin: IN STD_LOGIC;
                  s: OUT STD_LOGIC_VECTOR (length-1 DOWNTO 0);
                  cout: OUT STD_LOGIC);
```

```
END adder;

ARCHITECTURE adder OF adder IS
BEGIN
      PROCESS (a, b, cin)
      VARIABLE carry : STD_LOGIC_VECTOR (length DOWNTO 0);
      BEGIN
            carry(0) := cin;
            FOR i IN 0 TO length-1 LOOP
                  s(i) <= a(i) XOR b(i) XOR carry(i);
                  carry(i+1) := (a(i) AND b(i)) OR (a(i) AND
                  carry(i)) OR (b(i) AND carry(i));
            END LOOP;
                  cout <= carry(length);
      END PROCESS;
END adder;
```

Cách 2:

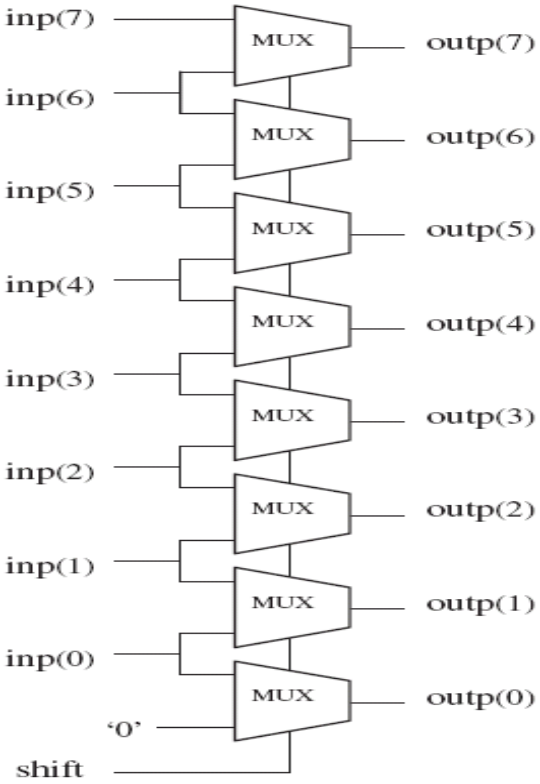
Dùng non-generic với các INTEGER

---- Solution 2: non-generic, with INTEGERS ----

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY adder IS
      PORT ( a, b: IN INTEGER RANGE 0 TO 255;
                  c0: IN STD_LOGIC;
                  s: OUT INTEGER RANGE 0 TO 255;
                  c8: OUT STD_LOGIC);
END adder;

ARCHITECTURE adder OF adder IS
BEGIN
      PROCESS (a, b, c0)
            VARIABLE temp : INTEGER RANGE 0 TO 511;
      BEGIN
            IF (c0='1') THEN temp:=1;
            ELSE temp:=0;
            END IF;
            temp := a + b + temp;
            IF (temp > 255) THEN
                  c8 <= '1';
                  temp := temp; ---256
            ELSE c8 <= '0';
            END IF;
            s <= temp;
      END PROCESS;
END adder;
```

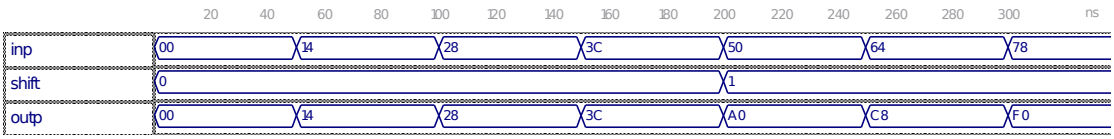
Ví dụ 6.6b:

Bộ dịch đơn giản: dịch vector đầu vào (kích thước 8) ‘0’ hoặc ‘1’ về phía trái. Khi dịch, bit LSB phải được điền ‘0’. Nếu shift = 0 thì outp = inp; nếu shift = 1 thì outp(0) = ‘0’ và outp(i) = inp(i-1) với 1 ≤ i ≤ 7.



Hình 6.6b.1. Bộ dich đơn giản

Kết quả mô phỏng:



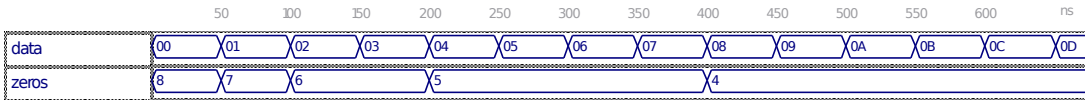
Hình 6.6b.2. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY barrel IS
      GENERIC (n: INTEGER := 8);
      PORT ( inp: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
                  shift: IN INTEGER RANGE 0 TO 1;
                  outp: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0));
END barrel;

ARCHITECTURE RTL OF barrel IS
BEGIN
      PROCESS (inp, shift)
      BEGIN
            IF (shift=0) THEN
```

```
                outp <= inp;
            ELSE
                outp(0) <= '0';
                FOR i IN 1 TO inp'HIGH LOOP
                    outp(i) <= inp(i-1);
                END LOOP;
            END IF;
      END PROCESS;
END RTL;
```

Ví dụ 6.6c:

Bộ đếm số số ‘0’ của một vector nhị phân, bắt đầu từ bên trái



Hình 6.6c.1. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY LeadingZeros IS
      PORT ( data: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                  zeros: OUT INTEGER RANGE 0 TO 8);
END LeadingZeros;

ARCHITECTURE behavior OF LeadingZeros IS
BEGIN
      PROCESS (data)
            VARIABLE count: INTEGER RANGE 0 TO 8;
      BEGIN
            count := 0;
            FOR i IN data'RANGE LOOP
                  CASE data(i) IS
                        WHEN '0' => count := count + 1;
                        WHEN OTHERS => EXIT;
                  END CASE;
            END LOOP;
            zeros <= count;
      END PROCESS;
END behavior;
```

6.7. Bad Clocking.

Trình biên dịch nói chung không có khả năng tổng hợp các mã chứa các phép gán cho tín hiệu giống nhau tại cả chuyển tiếp của tín hiệu đồng hồ (clock) tham chiếu (tại sườn dương cộng tại sườn âm). Trong trường hợp này, trình biên dịch có thể thông báo một thông điệp “signal does not hold value after clock edge” hoặc tương tự.

Ví dụ:

Bộ đếm phải được tăng tại mọi sự chuyển tiếp của tín hiệu clock (sườn dương cộng sườn dương)

```
PROCESS (clk)
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            counter <= counter + 1;
      ELSIF(clk'EVENT AND clk='0') THEN
            counter <= counter + 1;
      END IF;
      ...
END PROCESS;
```

Trình biên dịch có thể còn thông báo rằng tín hiệu counter bị nhân dồn. Trong trường hợp này, việc biên dịch sẽ bị treo.

Khía cạnh quan trọng khác là thuộc tính EVENT phải có liên quan tới điều kiện kiểm tra. Ví dụ: lệnh IF (clk'EVENT and clk='1') là đúng, nhưng chỉ sử dụng IF (clk'EVENT) có thể trình biên dịch giả sử một giá trị kiểm tra mặc định (and clk='1') hoặc thông báo một thông điệp "clock not locally stable".

```
PROCESS (clk)
BEGIN
      IF (clk'EVENT) THEN
            counter := counter + 1;
      END IF;
      ...
END PROCESS;
```

Khi PROCESS được chạy mọi thời điểm clk thay đổi, bộ đếm có thể được tăng hai trên mỗi chu kỳ clock. Tuy nhiên, điều này không xảy ra. Nếu trình biên dịch giả thiết một giá trị mặc định, một mạch lỗi sẽ được tổng hợp, bởi vì chỉ một sườn của clk sẽ được quan tâm; nếu không có giá trị mặc định giả thiết, thì một thông điệp lỗi và không có sự biên dịch được mong muốn.

Nếu một tín hiệu xuất hiện trong danh sách nhạy, nhưng không xuất hiện trong bất kỳ phép gán nào của PROCESS, thì xem như trình biên dịch sẽ bỏ qua. Tuy nhiên, một thông điệp "ignored unnecessary pin clk" có thể được thông báo.

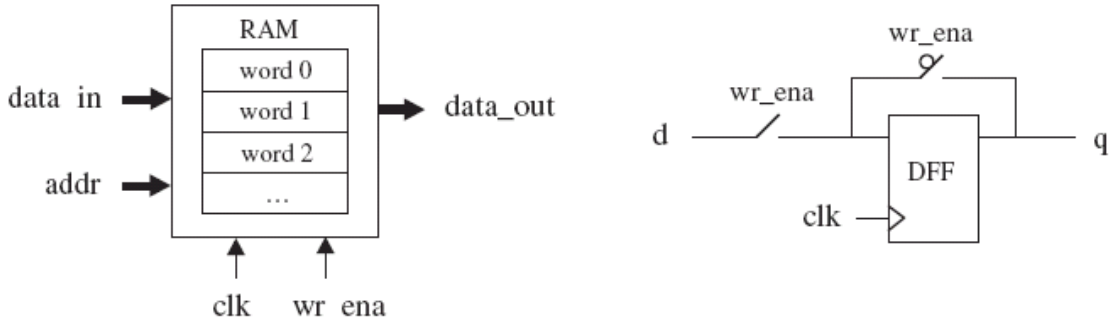
```
PROCESS (clk)
BEGIN
      counter := counter + 1;
      ...
END PROCESS;
```

Hai đoạn mã PROCESS sau sẽ được tổng hợp chính xác bởi bất kỳ trình biên dịch nào. Tuy nhiên, chú ý rằng sử dụng một tín hiệu khác nhau trong mỗi PROCESS.

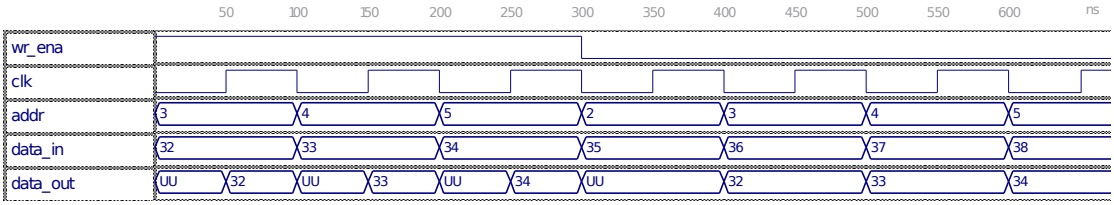
```
PROCESS (clk)
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            x <= d;
      END IF;
END PROCESS;

PROCESS (clk)
BEGIN
      IF (clk'EVENT AND clk='0') THEN
            y <= d;
      END IF;
END PROCESS;
```

Ví dụ 6.7a:

RAM (Random Acess Memory), dung lượng 16 từ nhớ x 8 bit



Hình 6.7a.1. RAM



Hình 6.7a.2. Kết quả mô phỏng

Mạch có bus dữ liệu vào (data_in), bus dữ liệu ra (data_out), bus địa chỉ (addr), cộng tín hiệu clock (clk) và các chân cho phép ghi (wr_ena). Khi wr_ena được xác nhận, tại sườn dương tiếp theo của clk, vector có mặt tại data_in phải được lưu trữ tại địa chỉ được mô tả bởi addr. Đầu ra, data_out, bằng cách xử lý khác, phải hiển thị liên tục dữ liệu chọn bởi addr.

Khi wr_ena ở mức thấp, q được nối với đầu vào của flip-flop, và d được mở, vì vậy không có dữ liệu mới sẽ được ghi vào bộ nhớ. Khi wr_ena trở về mức cao, d được nối với đầu vào của thanh ghi, vì vậy tại sườn dương tiếp theo của clk, d sẽ ghi đè giá trị liền trước.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
```

```
ENTITY ram IS
      GENERIC ( bits: INTEGER := 8;-- # of bits per word
                        words: INTEGER := 16);--#of words in the mem
      PORT ( wr_ena, clk: IN STD_LOGIC;
                  addr: IN INTEGER RANGE 0 TO words-1;
                  data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
                  data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END ram;

ARCHITECTURE ram OF ram IS
      TYPE vector_array IS ARRAY (0 TO words-1) OF
                  STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
      SIGNAL memory: vector_array;
BEGIN
      PROCESS (clk, wr_ena)
      BEGIN
            IF (wr_ena='1') THEN
                  IF (clk'EVENT AND clk='1') THEN
                        memory(addr) <= data_in;
                  END IF;
            END IF;
      END PROCESS;
      data_out <= memory(addr);
END ram;
```

6.8. Sử dụng mã tuần tự để thiết kế các mạch tổ hợp.

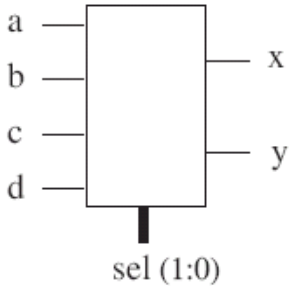
Mã tuần tự có thể được sử dụng để thực hiện các hệ dãy hay tổ hợp. Trong trường hợp hệ dãy, các thanh ghi là cần thiết, vì vậy sẽ được suy ra bởi trình biên dịch. Tuy nhiên, điều này sẽ không xảy ra trong trường hợp hệ tổ hợp. Hơn nữa, nếu mã được dùng cho hệ tổ hợp, thì bảng thật đầy đủ nên được môt tả rõ ràng trong mã.

Để thoả mãn các tiêu chuẩn trên có các luật được xét:

- Luật 1: Đảm bảo tất cả tín hiệu đầu vào sử dụng trong PROCESS xuất hiện trong danh sách nhạy của nó. Trình biên dịch đưa ra cảnh báo nếu một tín hiệu đầu vào đã cho không được chứa trong danh sách nhạy, và sau đó xử lý nếu tín hiệu đã được chứa.
- Luật 2: Đảm bảo tất cả tổ hợp các tín hiệu đầu vào/đầu ra được bao gồm trong mã, bảng thật đầy đủ của mạch có thể được chứa (điều này đúng với cả mã tuần tự và mã đồng thời). Các đặc tả không đầy đủ của các tín hiệu đầu ra có thể gây cho việc tổng hợp để suy ra các chốt để giữ các giá trị liền trước.

Ví dụ 6.8a:

Thiết kế mạch tổ hợp sai

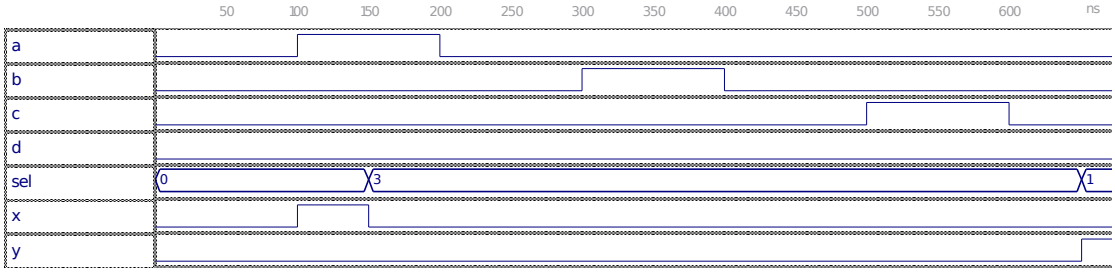


sel	x	y
00	a	0
01	b	1
10	c	
11	d	

sel	x	y
00	a	0
01	b	1
10	c	y
11	d	y

sel	x	y
00	a	0
01	b	1
10	c	X
11	d	X

Hình 6.8a.1. Mạch tổ hợp sai và các bảng thật



Hình 6.8a.2. Kết quả mô phỏng

x hoạt động như một bộ dồn kênh, y = 0 khi sel = “00”, hoặc = 1 nếu sel = “01”. Tuy nhiên các đặc tả được cung cấp cho y không dầy đủ.

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY example IS
      PORT (a, b, c, d: IN STD_LOGIC;
                  sel: IN INTEGER RANGE 0 TO 3;
                  x, y: OUT STD_LOGIC);
END example;

ARCHITECTURE example OF example IS
BEGIN
      PROCESS (a, b, c, d, sel)
      BEGIN
            IF (sel=0) THEN
                  x<=a;
                  y<='0';
            ELSIF (sel=1) THEN
                  x<=b;
                  y<='1';
            ELSIF (sel=2) THEN
                  x<=c;
            ELSE
                  x<=d;
            END IF;
      END PROCESS;
END example;
```

Sau khi biên dịch, các file báo cáo thể hiện không có flip-flop nào được suy ra. Giá trị giống nhau của đầu vào (sel = 3 = “11”), hai kết quả khác nhau cho y (khi sel =3 được đi trước sel = 0, kết quả y = ‘0’, trong khi y = ‘1’ khi sel = 3 được đi trước sel = 1).

y = (sel(0) AND sel(1)) OR (sel(0) AND y) OR (sel(1) AND y)

Do đó, một chốt (sử dụng các cổng AND/OR) đã được thực hiện, trả về bảng thật 2. Để tránh sử dụng chốt, nên sử dụng ‘X’ là giá trị không xác định, lệnh “y <= ‘X’;” phải được chứa dưới các dòng 22 và 24, do đó y = sel(0).

Chương 7: Signal và Variable

VHDL cung cấp hai đối tượng để giải quyết các giá trị dữ liệu không tĩnh (non-static): SIGNAL và VARIABLE. Nó còn cung cấp các cách để thiết lập các giá trị mặc định (static): CONSTANT và GENERIC.

CONSTANT và GENERIC có thể là toàn cục và có thể được sử dụng trong cả kiểu mã, đồng thời hoặc tuần tự. VARIABLE là cục bộ, chỉ có thể được sử dụng bên trong một phần của mã tuần tự (trong PROCESS, FUNCTION, hoặc PROCEDURE).

7.1. CONSTANT.

CONSTANT phục vụ cho việc thiết lập các giá trị mặc định.

Cú pháp:

```
CONSTANT name : type := value;
```

Ví dụ:

```
CONSTANT set_bit : BIT := '1';
CONSTANT datamemory : memory := (('0','0','0','0'),
                                 ('0','0','0','1'),
                                 ('0','0','1','1'));
```

CONSTANT có thể được khai báo trong PACKAGE, ENTITY và ARCHITECTURE. Khi khai báo trong gói (package), nó là toàn cục, gói có thể được sử dụng bởi nhiều thực thể (entity). Khi khai báo trong thực thể (sau PORT), nó là toàn cục với tất cả các kiến trúc (architecture) theo thực thể. Khi khai báo trong kiến trúc (trong phần khai báo của nó), nó chỉ toàn cục với mã của kiến trúc đó.

7.2. SIGNAL.

SIGNAL phục vụ giải quyết các giá trị vào và ra của mạch, cũng như là giữa các đơn vị bên trong của nó. Tín hiệu biểu diễn cho việc kết nối mạch (các dây). Thể hiện là, tất cả các PORT của ENTITY là các tín hiệu mặc định.

Cú pháp:

```
SIGNAL name : type [range] [:= initial_value];
```

Ví dụ:

```
SIGNAL control: BIT := '0';
SIGNAL count: INTEGER RANGE 0 TO 100;
SIGNAL y: STD_LOGIC_VECTOR (7 DOWNTO 0);
```

Khai báo của SIGNAL có thể được tạo ra ở các chỗ giống nhau như là khai báo CONSTANT.

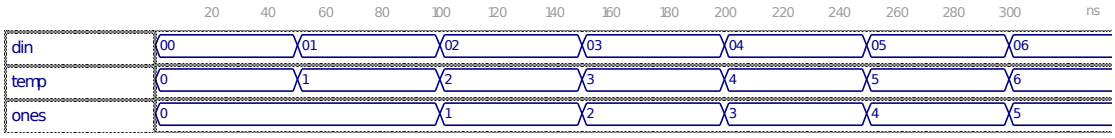
Khía cạnh quan trọng của SIGNAl, khi sử dụng bên trong một phần của mã tuần tự (PROCESS), sự cập nhật nó không tức thì. Giá trị mới của không nên được đợi để được đọc trước khi kết thúc PROCESS, FUNCTION, hoặc PROCEDURE tương ứng.

Phép toán gán cho SIGNAL là "<=" (count <= 35;). Giá trị khởi tạo không thể tổng hợp được, chỉ được xét khi mô phỏng.

Khía cạnh khác ảnh hưởng đến kết quả khi nhiều phép gán được tạo cùng SIGNAL. Trình biên dịch có thể thông báo và thoát sự tổng hợp, hoặc có thể suy ra mạch sai (bằng cách chỉ xét phép gán cuối cùng). Do đó, việc xét lập các giá trị khởi tạo, nên được thực hiện với VARIABLE.

Ví dụ 7.2a:

Bộ đếm số số '1' trong một vector nhị phân



Hình 7.2a.1. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY count_ones IS
      PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                  ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;

ARCHITECTURE not_ok OF count_ones IS
      SIGNAL temp: INTEGER RANGE 0 TO 8;
BEGIN
      PROCESS (din)
      BEGIN
            temp <= 0;
            FOR i IN 0 TO 7 LOOP
                  IF (din(i)='1') THEN
                        temp <= temp + 1;
                  END IF;
            END LOOP;
            ones <= temp;
      END PROCESS;
END not_ok;
```

7.3. VARIABLE

Ngược lại với CONSTANT và SIGNAL, VARIABLE chỉ biểu diễn thông tin cục bộ. Nó chỉ có thể được sử dụng bên trong PROCESS, FUNCTION, hay PROCEDURE (trong mã tuần tự). Việc cập nhật giá trị của nó là tức thì, vì vậy giá trị mới có thể được lập tức sử dụng trong dòng lệnh tiếp theo của mã.

Cú pháp:

```
VARIABLE name : type [range] [:= init_value];
```

Ví dụ:

```
VARIABLE control: BIT := '0';
VARIABLE count: INTEGER RANGE 0 TO 100;
VARIABLE y: STD_LOGIC_VECTOR (7 DOWNTO 0) := "10001000";
```

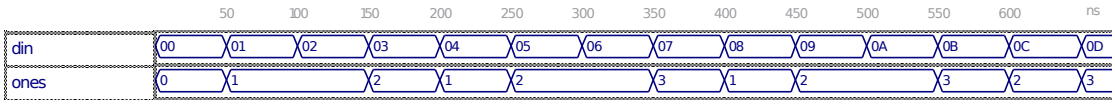
Khi VARIABLE chỉ có thể được sử dụng trong mã tuần tự, khai báo của nó chỉ có thể được thực hiện trong phần khai báo của PROCESS, FUNCTION, hay PROCEDURE.

Phép toán gán của VARIABLE là ":=" (count:=35;). Cũng giống như trường hợp của SIGNAl, giá trị khởi tạo không thể tổng hợp được, chỉ được xét khi mô phỏng.

Ví dụ 7.3a:

Bộ đếm số số '1' của một vector nhị phân

Khi cập nhật biến là tức thì, giá trị khởi tạo được thiết lập chính xác và không có thông báo nào về nhiều phép gán do trình biên dịch.



Hình 7.3a.1. Kết quả mô phỏng

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY count_ones IS
      PORT ( din: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
                  ones: OUT INTEGER RANGE 0 TO 8);
END count_ones;

ARCHITECTURE ok OF count_ones IS
BEGIN
      PROCESS (din)
            VARIABLE temp: INTEGER RANGE 0 TO 8;
```

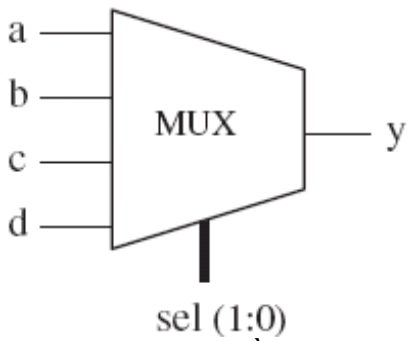
```
      BEGIN
            temp := 0;
            FOR i IN 0 TO 7 LOOP
                  IF (din(i)='1') THEN
                        temp := temp + 1;
                  END IF;
            END LOOP;
            ones <= temp;
      END PROCESS;
END ok;
```

	SIGNAL	VARIABLE
Phép gán	<=	:=
Tính năng	Biểu diễn sự kết nối các mạch (các dây)	Biểu diễn thông tin cục bộ
Phạm vi	Có thể là toàn cục (trên toàn bộ mã)	Cục bộ (chỉ trong PROCESS, FUNCTION, hay PROCEDURE tương ứng)
Hoạt động	Cập nhật không tức thì trong mã tuần tự (giá trị mới chỉ có thể dùng lúc kết thúc PROCESS, FUNCTION, hay PROCEDURE)	Cập nhật tức thì (giá trị mới có thể được sử dụng trong dòng lệnh tiếp theo của mã)
Sử dụng	Trong PACKAGE, ENTITY, hay ARCHITECTURE. Trong ENTITY, tất cả các PORT là các SIGNAL mặc định	Chỉ trong mã tuần tự, trong PROCESS, FUNCTION, hay PROCEDURE

Bảng 7.1. So sánh giữa SIGNAL và VARIABLE

Ví dụ 7.3b:

Bộ dồn kênh 4-1



Hình 7.3b.1. Bộ dồn kênh 4-1

Cách 1:

Sử dụng SIGNAL (không đúng)

-- Solution 1: using a SIGNAL (not ok) --

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
      PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                  y: OUT STD_LOGIC);
END mux;

ARCHITECTURE not_ok OF mux IS
      SIGNAL sel : INTEGER RANGE 0 TO 3;
BEGIN
      PROCESS (a, b, c, d, s0, s1)
      BEGIN
            sel <= 0;
            IF (s0='1') THEN sel <= sel + 1;
            END IF;
            IF (s1='1') THEN sel <= sel + 2;
            END IF;
            CASE sel IS
                  WHEN 0 => y<=a;
                  WHEN 1 => y<=b;
                  WHEN 2 => y<=c;
                  WHEN 3 => y<=d;
            END CASE;
      END PROCESS;
END not_ok;
```

Cách 2:

Sử dụng VARIABLE (đúng)

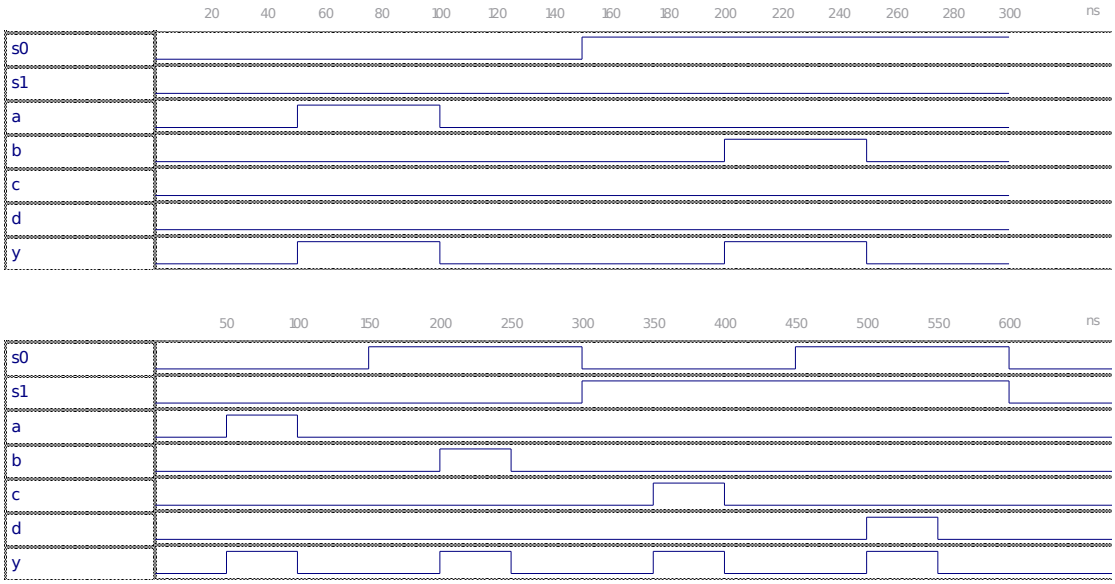
-- Solution 2: using a VARIABLE (ok) ----

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY mux IS
      PORT ( a, b, c, d, s0, s1: IN STD_LOGIC;
                  y: OUT STD_LOGIC);
END mux;

ARCHITECTURE ok OF mux IS
BEGIN
      PROCESS (a, b, c, d, s0, s1)
            VARIABLE sel : INTEGER RANGE 0 TO 3;
      BEGIN
            sel := 0;
            IF (s0='1') THEN sel := sel + 1;
```

```
            END IF;
            IF (s1='1') THEN sel := sel + 2;
            END IF;
            CASE sel IS
                  WHEN 0 => y<=a;
                  WHEN 1 => y<=b;
                  WHEN 2 => y<=c;
                  WHEN 3 => y<=d;
            END CASE;
      END PROCESS;
END ok;
```

Một lỗi thường xuyên khi sử dụng SIGNAL là không nhớ nó có thể yêu cầu một khoảng thời gian để cập nhật. Do đó, phép gán sel <= sel + 1 (dòng 16) trong cách 1, kết quả cộng thêm 1 bất kể giá trị vừa được tạo liền trước cho sel, với phép gán sel <= 0 (dòng 15) có thể không có thời gian để tạo. Điều này đúng với với sel <= sel +2 (dòng 18). Đây không là vấn đề khi sử dụng VARIABLE, phép gán của nó luôn tức thì.

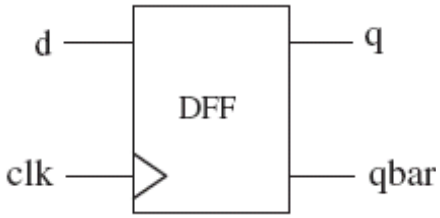
Khía cạnh thứ 2 có thể là một vấn đề trong cách 1 là hơn một phép toán đang được tạo cho cùng SIGNAL (sel, dòng 15, 16, và 18), có thể không được chấp nhận. Tóm lại, chỉ một phép gán với SIGNAL được phép bên trong PROCESS, vì vậy phần mềm chỉ xét phép gán cuối cùng (sel <= sel +2) hoặc đơn giản là đưa ra thông báo lỗi và kết thúc việc biên dịch. Đây cũng không bao giờ là vấn đề khi sử dụng VARIABLE.



Hình 7.3b.2. Kết quả mô phỏng cách 1 và 2

Ví dụ 7.3c:

DFF với q và qbar



Hình 7.3c.1. DFF

Cách 1:

Không đúng

---- Solution 1: not OK ----------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff IS
      PORT ( d, clk: IN STD_LOGIC;
                  q: BUFFER STD_LOGIC;
                  qbar: OUT STD_LOGIC);
END dff;

ARCHITECTURE not_ok OF dff IS
BEGIN
      PROCESS (clk)
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  q <= d;
                  qbar <= NOT q;
            END IF;
      END PROCESS;
END not_ok;
```

Cách 2:

Đúng

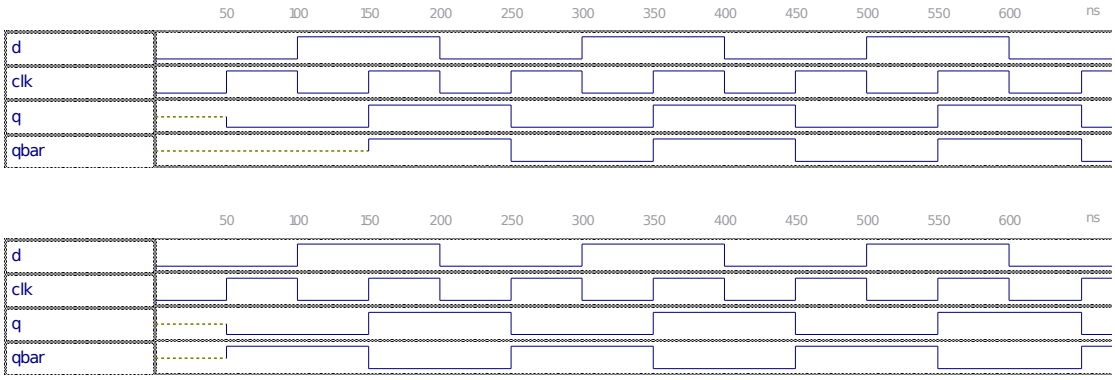
---- Solution 2: OK ----------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff IS
      PORT ( d, clk: IN STD_LOGIC;
                  q: BUFFER STD_LOGIC;
                  qbar: OUT STD_LOGIC);
END dff;

ARCHITECTURE ok OF dff IS
BEGIN
      PROCESS (clk)
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  q <= d;
            END IF;
      END PROCESS;
```

```
      qbar <= NOT q;
END ok;
```

Trong cách 1, các phép gán q<=d (dòng 16) và qbar<=NOT q (dòng 17) đều đồng bộ, vì vậy các giá trị mới của chúng sẽ chỉ được dùng lúc kết thúc PROCESS. Đây là vấn đề đối với qbar, bởi vì giá trị mới của q không vừa mới tạo ra. Do đó, qbar sẽ nhận giá trị đảo giá trị cũ của q. Giá trị đúng của qbar sẽ bị trễ một chu kỳ đồng hồ, gây cho mạch làm việc không chính xác.

Trong cách 2, thay qbar<=NOT q (dòng 30) bên ngoài PROCESS, do đó phép tính như một biểu thức đồng thời đúng.



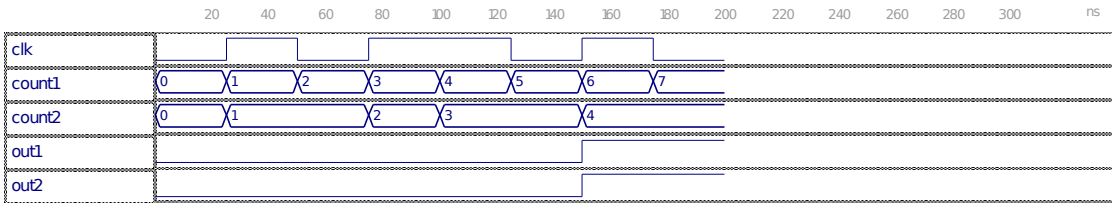
Hình 7.3c.2. Kết quả mô phỏng cách 1 và 2

Ví dụ 7.3d:

Bộ chia tần, chia tần số clock bởi 6.

Hình 7.3d.1. Bộ chia tần

Thực hiện hai đầu ra, một là dựa trên SIGNAL (count1), và một dựa trên VARIABLE (count2).



Hình 7.3d.2. Kết quả mô phỏng

```
LIBRARY ieee;
```

```
USE ieee.std_logic_1164.all;

ENTITY freq_divider IS
      PORT ( clk : IN STD_LOGIC;
                  out1, out2 : BUFFER STD_LOGIC);
END freq_divider;

ARCHITECTURE example OF freq_divider IS
      SIGNAL count1 : INTEGER RANGE 0 TO 7;
BEGIN
      PROCESS (clk)
            VARIABLE count2 : INTEGER RANGE 0 TO 7;
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  count1 <= count1 + 1;
                  count2 := count2 + 1;
                  IF (count1 = 7 ) THEN
                        out1 <= NOT out1;
                        count1 <= 0;
                  END IF;
                  IF (count2 = 7 ) THEN
                        out2 <= NOT out2;
                        count2 := 0;
                  END IF;
            END IF;
      END PROCESS;
END example;
```

7.4. Số thanh ghi.

Số flip-flop được suy ra từ mã bởi trình biên dịch. Mục đích là không chỉ hiểu tiếp cận yêu cầu số thanh ghi tối thiểu, mà còn đảm bảo đoạn mã thực hiện mạch mong muốn.

Một SIGNAL sinh một flip-flop bất cứ khi nào một phép gán được tạo ra tại sự chuyển tiếp của tín hiệu khác, khi một phép gán đồng bộ xảy ra. Phép gán đồng bộ, có thể chỉ xảy ra bên trong PROCESS, FUNCTION, hay PROCEDURE (thường là một khai báo kiểu "IF signal'EVENT …" hoặc "WAIT UNTIL …").

Một VARIABLE sẽ không sinh các flip-flop cần thiết nếu giá trị của nó không bao giờ rời PROCESS (hoặc FUNCTION, hoặc PROCEDURE). Tuy nhiên, nếu một giá trị được gán cho một biến tại sự chuyển tiếp của tín hiệu khác, và giá trị thậm chí được đưa tới một tín hiệu (rời PROCESS), thì các flip-flop sẽ được suy ra. Một VARIABLE còn sinh một thanh ghi khi nó được sử dụng trước một giá trị vừa được gán cho nó.

Ví dụ:

Trong PROCESS, output1 và output2 đều sẽ được lưu trữ (suy ra các flip-flop), bởi vì cả hai đều được gán tại sự chuyển tiếp của tín hiệu khác (clk).

```
PROCESS (clk)
```

```
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            output1 <= temp; -- output1 stored
            output2 <= a; -- output2 stored
      END IF;
END PROCESS;
```

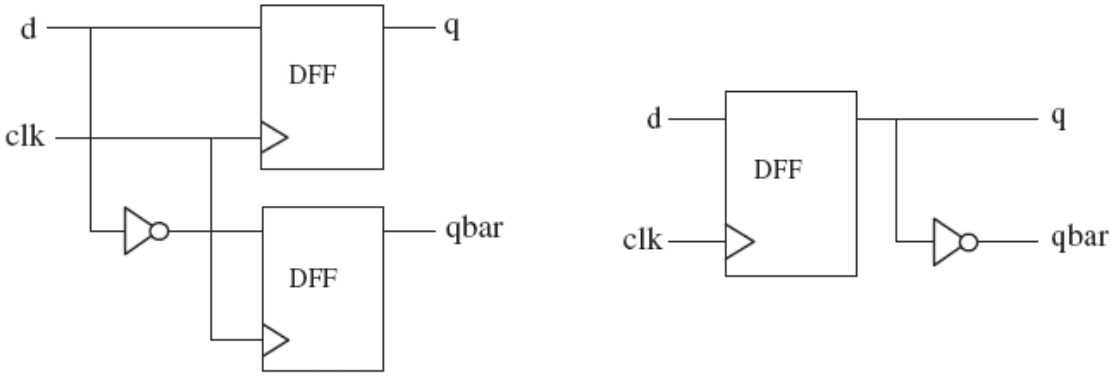
Trong PROCESS tiếp theo, chỉ output1 được lưu trữ (output2 sẽ tạo cách sử dụng các cổng logic).

```
PROCESS (clk)
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            output1 <= temp; -- output1 stored
      END IF;
            output2 <= a; -- output2 not stored
END PROCESS;

      Trong PROCESS, biến temp sẽ gây ra tín hiệu x để lưu trữ

PROCESS (clk)
VARIABLE temp: BIT;
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            temp <= a;
      END IF;
      x <= temp; -- temp causes x to be stored
END PROCESS;
```

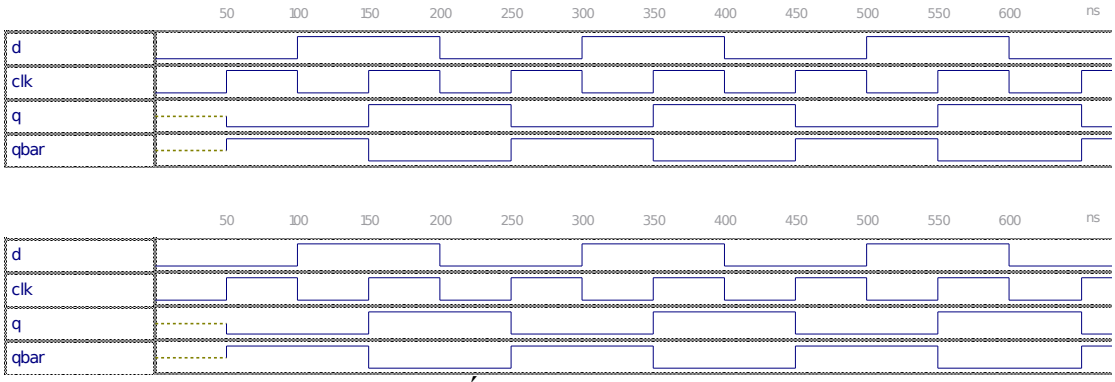
Ví dụ 7.4a:

DFF với q và qbar

Cách 1 có 2 phép gán SIGNAL đồng bộ (dòng 16-17), vì vậy 2 flip-flop sẽ được sinh. Cách 2 có một trong các phép gán là đồng bộ, việc tổng hợp sẽ luôn suy ra chỉ một flip-flop



Hình 7.4a.1. Các mạch suy ra từ mã của cách 1 và 2



Hình 7.4a.2. Kết quả mô phỏng cách 1 và 2

Cách 1:

Sinh hai DFF

---- Solution 1: Two DFFs ----------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff IS
      PORT ( d, clk: IN STD_LOGIC;
                  q: BUFFER STD_LOGIC;
                  qbar: OUT STD_LOGIC);
END dff;

ARCHITECTURE two_dff OF dff IS
BEGIN
      PROCESS (clk)
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  q <= d; -- generates a register
                  qbar <= NOT d; -- generates a register
            END IF;
      END PROCESS;
END two_dff;
```

Cách 2:

Sinh một DFF

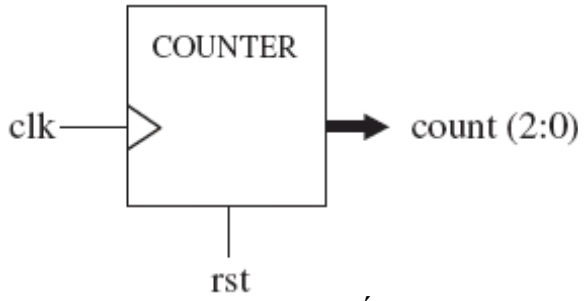
---- Solution 2: One DFF ----------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY dff IS
      PORT ( d, clk: IN STD_LOGIC;
                  q: BUFFER STD_LOGIC;
                  qbar: OUT STD_LOGIC);
END dff;

ARCHITECTURE one_dff OF dff IS
BEGIN
```

```
      PROCESS (clk)
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  q <= d; -- generates a register
            END IF;
      END PROCESS;
      qbar <= NOT q; -- uses logic gate (no register)
END one_dff;
```

Ví dụ 7.4b:

Bộ đếm 0 - 7



Hình 7.4b.1. Bộ đếm 0 – 7

Cách 1:

Một phép gán VARIABLE đồng bộ được tao ra (dòng 14-15). Một VARIABLE có thể sinh các thanh ghi bởi vì phép gán của nó (dòng 15) tại sự chuyển tiếp của tín hiệu khác (clk, dòng 14) và giá trị của nó không rời PROCESS (dòng 17).

```
------ Solution 1: With a VARIABLE --------
ENTITY counter IS
PORT ( clk, rst: IN BIT;
count: OUT INTEGER RANGE 0 TO 7);
END counter;

ARCHITECTURE counter OF counter IS
BEGIN
      PROCESS (clk, rst)
            VARIABLE temp: INTEGER RANGE 0 TO 7;
      BEGIN
            IF (rst='1') THEN
                  temp:=0;
            ELSIF (clk'EVENT AND clk='1') THEN
                  temp := temp+1;
            END IF;
            count <= temp;
```

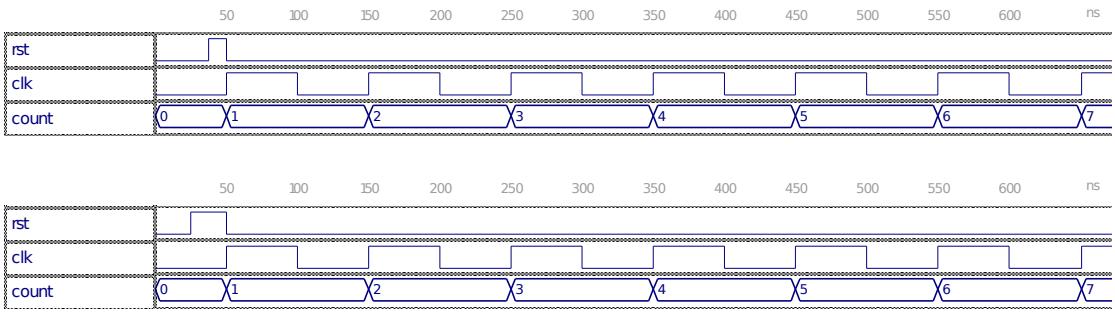
```
      END PROCESS;
END counter;
```

Cách 2:

Một phép gán SIGNAL đồng bộ xảy ra (dòng 13-14). Chỉ sử dụng các SIGNAL. Chú ý, khi không có tín hiệu phụ được sử dụng, count cần được khai báo như kiểu BUFFER (dòng 14), bởi vì nó được gán một giá trị và cũng được đọc (sử dụng) nội tại (dòng 14). Một SIGNAL, giống như một VARIABLE, có thể cũng được tăng khi sử dụng trong mã tuần tự.

```
------ Solution 2: With SIGNALS only -------
ENTITY counter IS
      PORT ( clk, rst: IN BIT;
                  count: BUFFER INTEGER RANGE 0 TO 7);
END counter;

ARCHITECTURE counter OF counter IS
BEGIN
      PROCESS (clk, rst)
      BEGIN
            IF (rst='1') THEN
                  count <= 0;
            ELSIF (clk'EVENT AND clk='1') THEN
                  count <= count + 1;
            END IF;
      END PROCESS;
END counter;
```

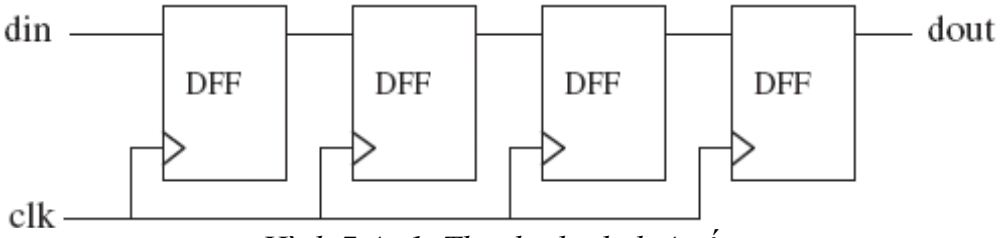
Từ 2 cách trên, 3 flip-flop được suy ra (để giữ 3 bit tín hiệu đầu ra count).



Hình 7.4b.2. Kết quả mô phỏng cách 1 và 2

Ví dụ 7.4c:

Thanh ghi dịch 4 cấp



Hình 7.4c.1. Thanh ghi dịch 4 cấp

Cách 1:

3 VARIABLE được sử dụng (a, b, và c, dòng 10). Tuy nhiên các biến được sử dụng trước các giá trị được gán cho chúng (đảo ngược thứ tự, bắt đầu với dout, dòng 13, và kết thúc với din, dòng 16). Kết quả là, các flip-flop sẽ được suy ra, lưu trữ các giá trị từ phép chạy liền trước của PROCESS.

-------- Solution 1: ----------

```
ENTITY shift IS
PORT ( din, clk: IN BIT;
dout: OUT BIT);
END shift;

ARCHITECTURE shift OF shift IS
BEGIN
      PROCESS (clk)
            VARIABLE a, b, c: BIT;
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  dout <= c;
                  c := b;
                  b := a;
                  a := din;
            END IF;
      END PROCESS;
END shift;
```

Cách 2:

Các biến được thay thế bởi các SIGNAL (dòng 8), và các phép gán được tạo ra trong thứ tự trực tiếp (từ din-dout, dòng 13-16). Khi các phép gán tín hiệu tại sự chuyển tiếp tín hiệu khác sinh các thanh ghi, mạch đúng sẽ được suy ra.

-------- Solution 2: ----------

```
ENTITY shift IS
      PORT ( din, clk: IN BIT;
                  dout: OUT BIT);
END shift;

ARCHITECTURE shift OF shift IS
      SIGNAL a, b, c: BIT;
BEGIN
      PROCESS (clk)
```

```
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  a <= din;
                  b <= a;
                  c <= b;
                  dout <= c;
            END IF;
      END PROCESS;
END shift;
```

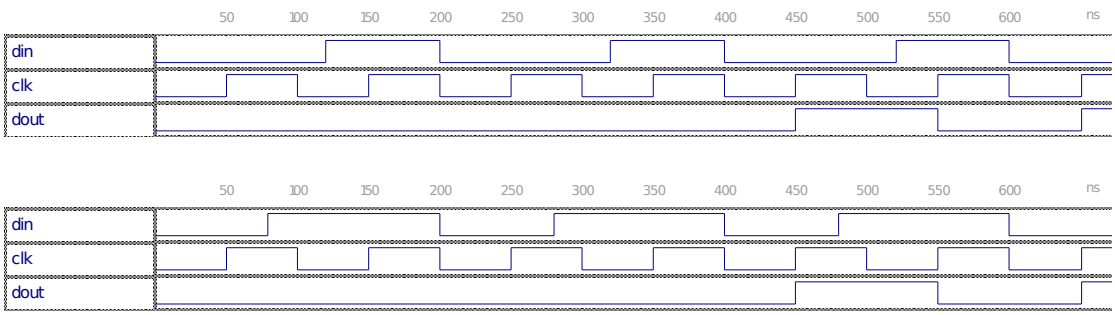
Cách 3:

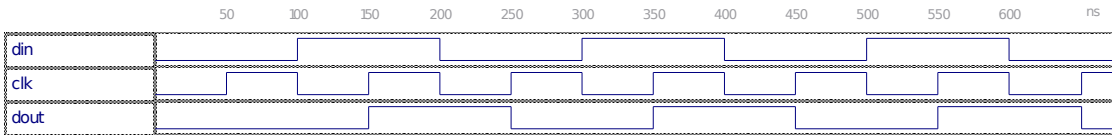
Các biến giống nhau của cách 1 đã bị chiếm, nhưng trong thứ tự trực tiếp (từ din-dout, dòng 13-16). Tuy nhiên, một phép gán cho một biến là tức thì, và khi các biến đang được sử dụng trong thứ tự trực tiếp (sau khi các giá trị vừa được gán cho chúng), dòng 13-15 thành 1 dòng, tương đương với c:=din. Giá trị của c rời PROCESS trong dòng tiếp theo (dòng 16), khi một phép gán tín hiệu (dout <= c) xảy ra tại sự chuyển tiếp của clk. Do đó, một thanh ghi sẽ được suy ra từ cách 3, nên không tạo kết quả mạch chính xác.

-------- Solution 3: ----------

```
ENTITY shift IS
      PORT ( din, clk: IN BIT;
                  dout: OUT BIT);
END shift;

ARCHITECTURE shift OF shift IS
BEGIN
      PROCESS (clk)
            VARIABLE a, b, c: BIT;
      BEGIN
            IF (clk'EVENT AND clk='1') THEN
                  a := din;
                  b := a;
                  c := b;
                  dout <= c;
            END IF;
      END PROCESS;
END shift;
```

Đầu ra dout là bốn sườn clock dương sau đầu vào din ở cách 1, nhưng chỉ một sườn dương sau đầu vào ở cách 2.

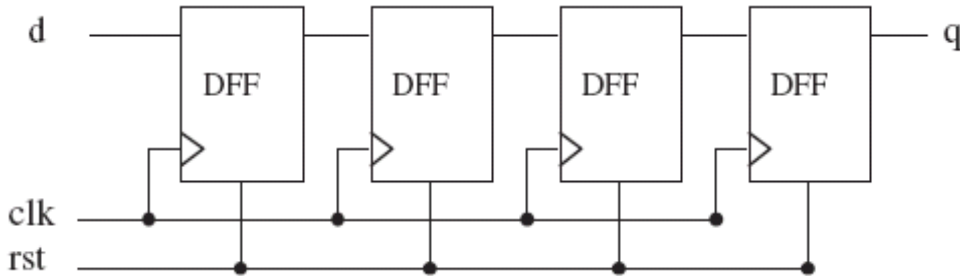




Hình 7.4c.2. Kết quả mô phỏng cách 1, 2, và 3

Ví dụ 7.4d:

Thanh ghi dịch 4 bit



Hình 7.4d.1. Thanh ghi dịch 4 bit

Bit ra (q) phải là 4 sườn clock dương sau bit vào (d). Reset phải là không đồng bộ, xoá tất cả các đầu ra flip-flop về ‘0’ khi kích hoạt.

Cách 1:

Sử dụng một SIGNAL để sinh các flip-flop. Các thanh ghi được tạo bởi vì một phép gán cho một tín hiệu được tạo ra tại sự chuyển tiếp của tín hiệu khác (dòng 17-18).

---- Solution 1: With an internal SIGNAL ---

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shiftreg IS
      PORT ( d, clk, rst: IN STD_LOGIC;
                  q: OUT STD_LOGIC);
END shiftreg;

ARCHITECTURE behavior OF shiftreg IS
      SIGNAL internal: STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
      PROCESS (clk, rst)
      BEGIN
            IF (rst='1') THEN
                  internal <= (OTHERS => '0');
            ELSIF (clk'EVENT AND clk='1') THEN
                  internal <= d & internal(3 DOWNTO 1);
            END IF;
      END PROCESS;
      q <= internal(0);
```

```
END behavior;
```

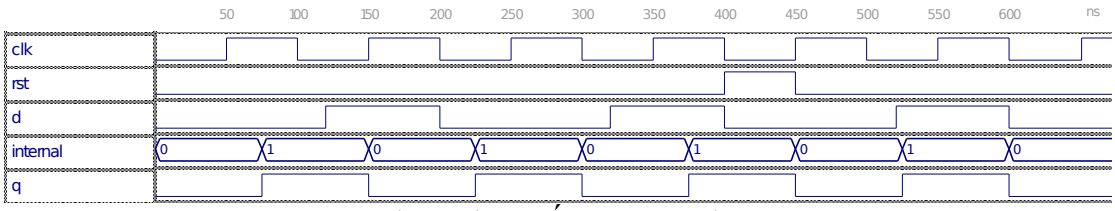
Cách 2:

Sử dụng một VARIABLE. Phép gán tại sự chuyển tiếp của tín hiệu khác được tạo ra cho một biến (dòng 17-18), nhưng khi giá trị của nó rời PROCESS (nó được chuyển đến một port trong dòng 20), nó cũng suy ra các thanh ghi.

-- Solution 2: With an internal VARIABLE ---

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;

ENTITY shiftreg IS
      PORT ( d, clk, rst: IN STD_LOGIC;
                  q: OUT STD_LOGIC);
END shiftreg;

ARCHITECTURE behavior OF shiftreg IS
BEGIN
      PROCESS (clk, rst)
            VARIABLE internal: STD_LOGIC_VECTOR (3 DOWNTO 0);
      BEGIN
            IF (rst='1') THEN
                  internal := (OTHERS => '0');
            ELSIF (clk'EVENT AND clk='1') THEN
                  internal := d & internal(3 DOWNTO 1);
            END IF;
            q <= internal(0);
      END PROCESS;
END behavior;
```

Các mạch được tổng hợp là giống nhau (4 flip-flop được suy ra).



Hình 7.4d.2. Kết quả mô phỏng

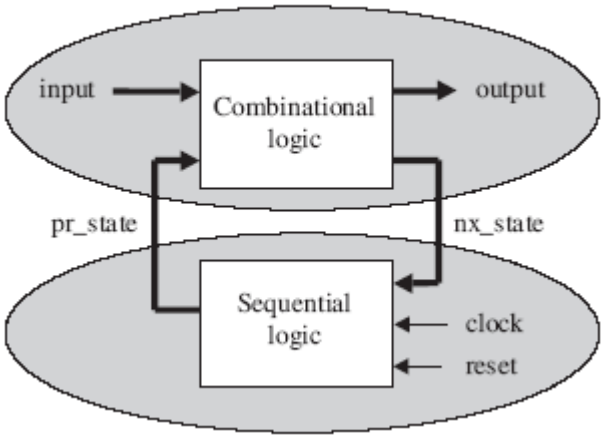
Chương 8: Máy trạng thái

Một thiết kế mạch số có thể được chia làm 2 thành phần: bộ xử lý dữ liệu và bộ điều khiển. Mối quan hệ giữa bộ điều khiển và bộ xử lý dữ liệu trong mạch được biểu diễn

Máy trạng thái hữu hạn (FSM) là một công nghệ mô hình hoá đặc biệt cho các mạch logic tuần tự. Mô hình đó có thể rất được giúp đỡ trong thiết kế của những loại hệ thống nào đó, đặc biệt là các thao tác của những hệ thống đó theo khuôn dạng tuần tự hoàn toàn xác định.

8.1. Giới thiệu.

Hình sau đây chỉ ra sơ đồ khối của một máy trạng thái một pha. Trong hình này, phần mạch dãy chứa các mạch dãy (flip-flops), phần cao chứa mạch logic tổ hợp.



Hình 8.1 Sơ đồ máy trạng thái

Phần mạch tổ hợp có 2 đầu vào và 2 đầu ra:

+ Đầu vào thứ nhất: là đầu vào trạng thái hiện tại của máy.
+ Đầu vào thứ 2: là đầu vào từ bên ngoài.
+ Đầu ra thứ nhất: là đầu ra phía ngoài
+ Đầu ra thứ 2: là trạng thái tiếp theo của máy.

Phần mạch dãy có:

+ 3 đầu vào: clock, reset, và trạng thái tiếp theo
+ 1 đầu ra: trạng thái hiện tại.

Tất cả các flip-flop đều nằm trong phần này, các tín hiệu clock và reset phải được kết nối với các flip – flop để thực hiện việc điều khiển.

Như vậy, một máy ôtômát hữu hạn là một bộ 6 thông số <X, Y, S, s_0, δ,λ >, trong đó:

- X - Tập hợp các tín hiệu vào của ôtômat:
  $X = \{ x_1(t),\ldots,x_n(t)\}$
- Tập các tín hiệu ra của ôtômat:
  $Y = \{y_1(t),\ldots,y_m(t)\}$
- Tập hợp các trạng thái của ôtômat:
  $S = \{s_1(t),\ldots,s_s(t)\}$
- Hàm δ(s, x) – hàm chuyển trạng thái của ôtômat
- Hàm λ(s,x) – hàm đầu ra của ôtômat.

Tương ứng với các phương pháp tính toán hàm chuyển trạng thái và hàm ra, chúng ta có các loại ôtômat khác nhau. Hai dạng ôtômat hữu hạn chuyên dụng là: ôtômat Moore và ôtômat Mealy.

Quay lại với hình vẽ trên, mạch cần thiết kế được chia làm hai đoạn. Việc chia đoạn như thế này sẽ giúp chúng ta thiết kế tốt hơn. Chúng ta sẽ thiết kế 2 phần theo những cách khác nhau. Cụ thể trong môi trường VHDL, phần mạch dãy chúng ta sẽ thực hiện trong PROCESS và phần mạch tổ hợp chúng ta

có thể thực hiện theo cấu trúc hoặc tuần tự hoặc kết hợp cả cấu trúc lẫn tuần tự. Tuy nhiên mã tuần tự có thể áp dụng cho cả 2 loại logic: tổ hợp và tuần tự.

Thông thường các tín hiệu clock và các tín hiệu reset trong phần mạch dãy sẽ xuất hiện trong PROCESS (trừ khi tín hiệu reset là đồng bộ hoặc không được sử dụng, tín hiệu WAIT được sử dụng thay cho lệnh IF). Khi tín hiệu reset được xác nhận, trạng thái hiện tại sẽ được thiết lập cho trạng thái khởi tạo của hệ thống. Mặt khác, tại sườn đồng hồ thực tế, các flip-flop sẽ lưu trữ trạng thái tiếp theo, do đó sẽ chuyển nó tới đầu ra của phần mạch dãy (trạng thái hiện tại).

Một điều quan trọng liên quan tới phương pháp FSM là : về nguyên tắc chung là bất kỳ một mạch dãy nào cũng có thể được mô hình hoá thành 1 máy trạng thái, nhưng điều này không phải luôn luôn thuận lợi. Vì có nhiều trường hợp (đặc biệt là các mạch thanh ghi như: bộ đếm,…) nếu thiết kế theo phương pháp FSM thì mã nguồn có thể trở nên dài hơn, phức tạp hơn, mắc nhiều lỗi hơn so với phương pháp thông thường.

Như thành một quy tắc nhỏ, phương pháp FSM thì thích hợp với các hệ thống mà thao tác của nó là một dãy hoàn toàn được cấu trúc, ví dụ: các mạch điều khiển số. Vì đối với các hệ thống loại này thì tất cả các trạng thái của nó có thể dễ dàng được liệt kê. Khi soạn thảo mã VHDL, thì các trạng thái này sẽ được khai báo trong phần đầu của phần ARCHITECTURE dưới dạng kiểu dữ liệu liệt kê được định nghĩa bởi người sử dụng.

8.2. Thiết kế theo kiểu 1 (thiết kế theo mô hình may moore).

Có vài phương pháp có thể được hình thành để thiết kế một FSM. Chúng ta sẽ mô tả chi tiết một ví dụ mẫu mà mạch hoàn toàn được cấu trúc và dễ dàng áp dụng. Trong đó phần mạch dãy của máy trạng thái sẽ tách biệt với phần mạch tổ hợpcủa nó (hình vẽ trên).

Tất cả các trạng thái của máy luôn luôn được khai báo rõ ràng bằng cách sử dụng kiểu dữ liệu liệt kê.

Thiết kế phần mạch dãy:

Trên hình trên, các flip-flop nằm ở phần mạch dãy. Các đầu vào từ bên ngoài của phần này là các tín hiệu clock và reset. Các tín hiệu này được nối với các Flip-flop. Một đầu vào khác (bên trong) là trạng thái tiếp theo. Đầu ra duy nhất của phần này là trang thái hiện tại. Để xây dựng cho phần mạch dãy này, ta cần sử dụng cấu trúc PROCESS. Trong cấu trúc của PROCESS chúng ta co thể sẽ sử dụng các câu lệnh tuần tự như lệnh IF, WAIT, CASE, LOOP.

Khuôn mẫu thiết kế của phần mạch dãy sẽ như sau:

```
PROCESS (reset, clock)
BEGIN
      IF reset = '1' THEN
            Trang_thai_hien_tai <= Trang_thai_0 ;
      ELSIF (clock 'EVENT and clock = '')  THEN
            Trang_thai_hien_tai                    <=
Trang_thai_tiep_theo;
      END IF ;
END PROCESS ;
```

Mã chỉ ra ở đây là rất đơn giản. Nó chỉ chứa một tín hiệu reset đồng bộ. Tín hiệu reset này sẽ xác định trạng thái khởi đầu của hệ thống, sau đó là lưu trữ đồng bộ trạng thái tiếp theo (tại sườn dương đông hồ),và đưa ra đầu ra của phần mạch dãy trạng thái hiện tại.

Việc thiết kế cho phần mạch dãy này thì đơn giản vì nó là một chuẩn cơ bản, và số lượng các thanh ghi là tối thiểu. Ở phần 7.5, chúng ta biết rằng số lượng các flip – flop sẽ tính dựa vào số bits cần thiết để mã hoá tất cả các trạng thái của FSM. Bởi vậy nếu mẫu được mã hoá theo cách mặc định (mã hoá nhị phân) thì, chúng ta sẽ cần $\log_2 n$ Flip-flop, với n là số trạng thái.

Thiết kế phần mạch tổ hợp:

Ở hình 1, thì phần mạch tổ hợp là đầy đủ, vì vậy mã của nó sẽ không cần thiết theo tuần tự. Tốt nhất, chúng ta nên sử dụng mã đồng thời. Song trong ví dụ mẫu dưới đây chúng ta sẽ sử dụng mã tuần tự với câu lệnh CASE đóng vai trò trung tâm.

```
PROCESS (input, pr_state)
BEGIN
      CASE pr_state IS
            WHEN state0 =>
                  IF (input = ...) THEN
                        output <= <value>;
                        nx_state <= state1;
                  ELSE ...
                  END IF;
            WHEN state1 =>
                  IF (input = ...) THEN
                        output <= <value>;
                        nx_state <= state2;
                  ELSE ...
                  END IF;
            WHEN state2 =>
                  IF (input = ...) THEN
                        output <= <value>;
                        nx_state <= state2;
                  ELSE ...
                  END IF;
            ...
      END CASE;
END PROCESS;
```

Đoạn mã ở đây cũng rất đơn giản, và nó sẽ thực hiện 2 công việc chính:

+ Gán giá trị cho đầu ra.
+ Thiết lập trang thái tiếp theo.

Mẫu máy trạng thái cho kiểu thiết kế 1:

Dưới đây là khuôn mẫu hoàn chỉnh về kiểu thiết kế 1:

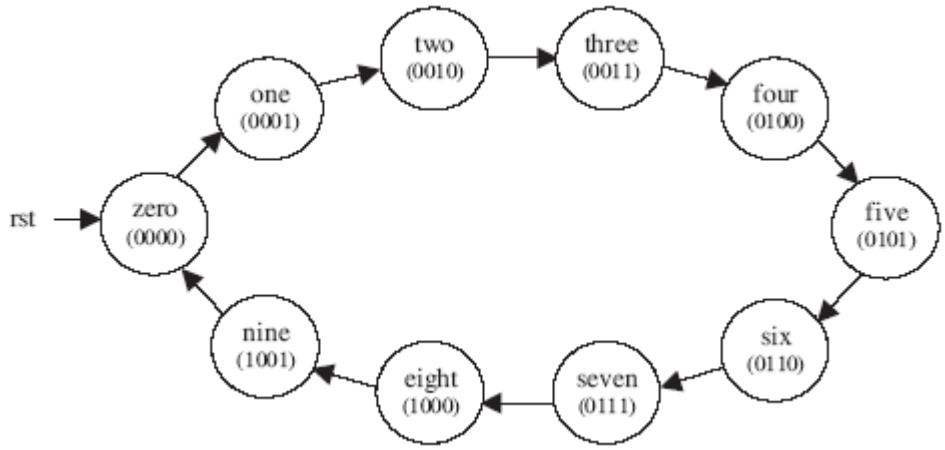
```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------------
ENTITY <entity_name> IS
PORT ( input: IN <data_type>;
     reset, clock: IN STD_LOGIC;
     output: OUT <data_type>);
END <entity_name>;
-----------------------------------------------------
ARCHITECTURE <arch_name> OF <entity_name> IS
TYPE state IS (state0, state1, state2, state3, ...);
SIGNAL pr_state, nx_state: state;
BEGIN
---------- Phần mạch dãy: ------------------------
PROCESS (reset, clock)
BEGIN
     IF (reset='1') THEN
          pr_state <= state0;
     ELSIF (clock'EVENT AND clock='1') THEN
          pr_state <= nx_state;
     END IF;
END PROCESS;
---------Phần mạch tổ hợp: ------------------------
PROCESS (input, pr_state)
BEGIN
     CASE pr_state IS
          WHEN state0 =>
               IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state1;
               ELSE ...
               END IF;
          WHEN state1 =>
               IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state2;
               ELSE ...
               END IF;
          WHEN state2 =>
               IF (input = ...) THEN
                    output <= <value>;
                    nx_state <= state3;
               ELSE ...
               END IF;
          ...
```

```
     END CASE;
END PROCESS;
END <arch_name>;
```

Ví dụ 8.1: Bộ đếm BCD

Một bộ đếm là một ví dụ của máy Moore, đầu ra chỉ phụ thuộc vào kết quả của trạng thái hiện tại. Giống như một mạch thanh ghi và một mạch dãy đơn giản. Để thiết kế mạch này, chúng ta có thể dùng phương pháp thông thường như những phần mạch mạch tổ hợp, nhưng ở đây ta sẽ dùng phương pháp FSM.

Giả sử ta cần thiết kế bộ đếm modul 10. Như vậy chúng ta sẽ cần có một may có 10 trang thái. Các trạng thái ở đây được gọi là zero, one,…,nine. Đồ hình trạng thái của máy được cho như sau:



Hình 8.2. Sơ đồ trạng thái của bộ đếm BCD

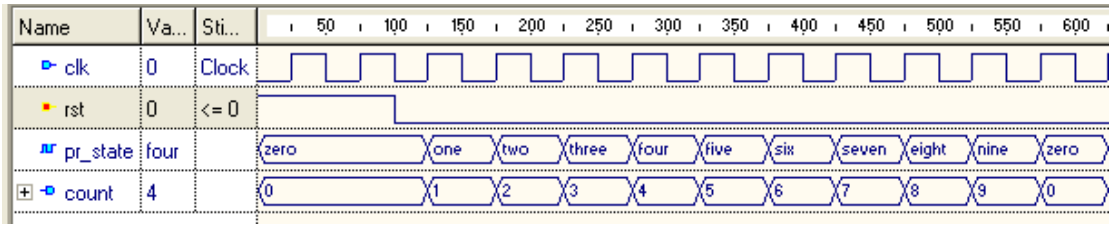
Mã VHDL cũng giống như khuôn mẫu của thiết kế mẫu 1. Trong đó: kiểu dữ liệu liệt kê sẽ xuất hiện ở dòng 11 – 12, thiết kế của phần mạch dãy sẽ từ dòng 16 đến dong 23, thiết kế của phần mạch tổ hợp(mạch tổ hợp) sẽ xuất hiện từ dòng 25 – 29. Do có 10 trang thái nên số lượng cac thanh ghi bằng là $[\log_2 10]= 4$.

Mã thiết kế sẽ như sau:

```
-----------------------------------------------
 LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 -------------------------------------------------
 ENTITY counterBCD IS
     PORT ( clk, rst: IN STD_LOGIC;
                count: OUT STD_LOGIC_VECTOR (3 DOWNTO 0));
 END counterBCD;
 -------------------------------------------------
 ARCHITECTURE state_machine OF counterBCD IS
```

```
TYPE state IS (zero, one, two, three, four,
                    five, six, seven, eight, nine);
SIGNAL pr_state, nx_state: state;
BEGIN
------------ Phan mach day: -----------------
PROCESS (rst, clk)
BEGIN
     IF (rst='1') THEN
          pr_state <= zero;
     ELSIF (clk'EVENT AND clk='1') THEN
          pr_state <= nx_state;
     END IF;
END PROCESS;
------------ Phan mach to hop: -----------------
PROCESS (pr_state)
BEGIN
     CASE pr_state IS
          WHEN zero =>
               count <= "0000";
               nx_state <= one;
          WHEN one =>
               count <= "0001";
               nx_state <= two;
          WHEN two =>
               count <= "0010";
               nx_state <= three;
          WHEN three =>
               count <= "0011";
               nx_state <= four;
          WHEN four =>
               count <= "0100";
               nx_state <= five;
          WHEN five =>
               count <= "0101";
               nx_state <= six;
          WHEN six =>
               count <= "0110";
               nx_state <= seven;
          WHEN seven =>
               count <= "0111";
               nx_state <= eight;
          WHEN eight =>
               count <= "1000";
               nx_state <= nine;
          WHEN nine =>
               count <= "1001";
               nx_state <= zero;
          END CASE;
END PROCESS;
END state_machine;
-------------------------------------------------
```

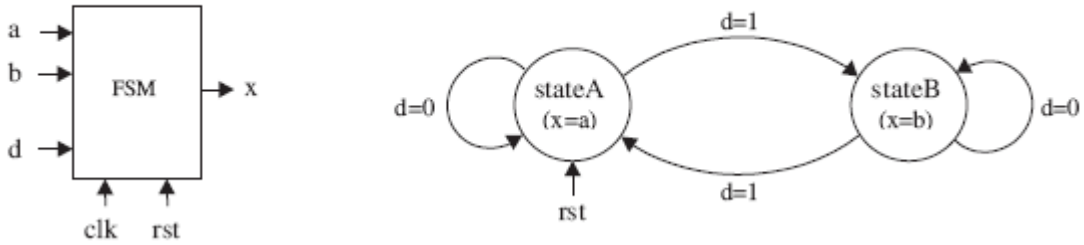
Mô phỏng kết quả:



Hình 8.3. Kết quả mô phỏng của bộ đếm BCD

Ví dụ 8.2: Máy trạng thái kết thúc kiểu 1

Hình 4 là sơ đồ khối của 1 FSM đơn giản. Hệ thống có 2 trạng thái: trạng thái A và trạng thái B. Máy phải chuyển trạng thái khi nhận được d = 1 và đầu ra mong muốn là x = a khi máy ở trạng thái A hoặc x = b khi máy ở trạng thái B.



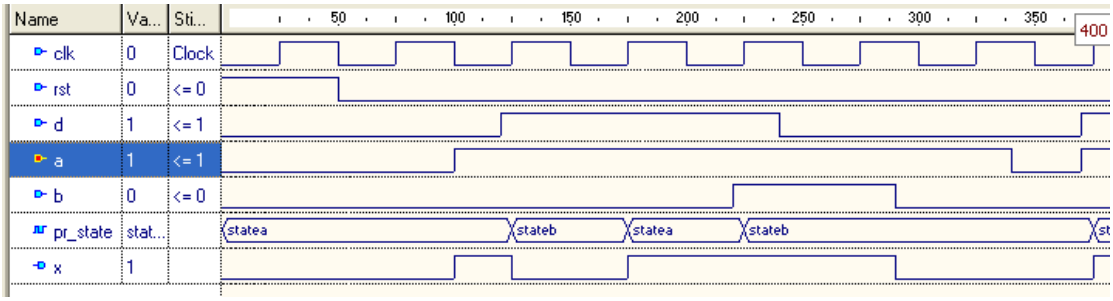
Hình 8.4. Máy trạng thái của ví dụ 8.2

Mã thiết kế sẽ như sau:

```
-------------------------------------------------
 LIBRARY ieee;
 USE ieee.std_logic_1164.all;
 -----------------------------------------------
 ENTITY vd_FSM IS
       PORT ( a, b, d, clk, rst: IN BIT;
                   x: OUT BIT);
 END vd_FSM;
 -----------------------------------------------
 ARCHITECTURE state_machine OF vd_FSM IS
      TYPE state IS (stateA, stateB);
      SIGNAL pr_state, nx_state: state;
 BEGIN
 ---------- Phan mach day: ---------------------
 PROCESS (rst, clk)
 BEGIN
      IF (rst='1') THEN
            pr_state <= stateA;
      ELSIF (clk'EVENT AND clk='1') THEN
```

```
            pr_state <= nx_state;
      END IF;
END PROCESS;
---------- Phan mach to hop: -----------------
PROCESS (a, b, d, pr_state)
BEGIN
      CASE pr_state IS
            WHEN stateA =>
                  x <= a;
                  IF (d='1') THEN nx_state <= stateB;
                  ELSE nx_state <= stateA;
                  END IF;
            WHEN stateB =>
                  x <= b;
                  IF (d='1') THEN nx_state <= stateA;
                  ELSE nx_state <= stateB;
                  END IF;
      END CASE;
END PROCESS;
END state_machine;
-----------------------------------------------
```

Kết quả mô phỏng:

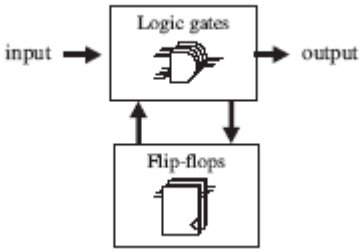


Hình 8.5. Kết quả mô phỏng cho ví dụ 8.2

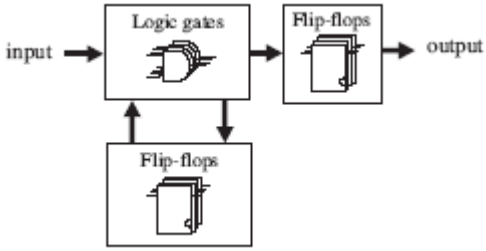
8.3. Thiết kế kiểu 2.

Như chúng ta thấy trong kiểu thiết kế 1 thì chỉ có trạng thái hiện tại được lưu trữ. Tất cả các mạch như vậy sẽ được tóm tắt như trong hình 8.6.1. Trong trường hợp này nếu mạch là máy Mealy (đầu ra của nó phụ thuộc vào đầu vào hiện tại), đầu ra có thể thay đổi khi đầu vào thay đổi (đầu ra không đồng bộ).

Trong nhiều ứng dụng, tín hiệu được yêu cầu là đồng bộ, thì đầu ra sẽ chỉ cập nhật khi thay đổi sườn clock. Để tạo ra máy đồng bộ Mealy, đầu ra phải được lưu trữ tốt, như trong hình 8.6.2



Hình 8.6.1 Sơ đồ mạch kiểu 1



Hình 8.6.2. Sơ đồ mạch kiểu 2

Cấu trúc như trong hình 8.6.2 sẽ là đối tượng của thiết kế kiểu 2.

Để thực hiện cấu trúc mới này, chúng ta cần có vài sự thay đổi so với thiết kế kiểu 1. Ví dụ, chúng ta có thể sử dụng một tín hiệu thêm (như tín hiệu trung gian) để tính toán giá trị đầu ra (đoạn trên), nhưng chỉ chuyển các giá trị của nó thành tín hiệu đầu ra khi sự kiện clock thay đổi (phần mạch dãy). Sự thay đổi này chúng ta sẽ thấy trong khuôn mẫu chỉ ra dưới đây:

Khuôn mẫu máy trạng thái của thiết kế 2

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY <ent_name> IS
      PORT (input: IN <data_type>;
            reset, clock: IN STD_LOGIC;
            output: OUT <data_type>);
END <ent_name>;
-------------------------------------------------------
ARCHITECTURE <arch_name> OF <ent_name> IS
TYPE states IS (state0, state1, state2, state3, ...);
SIGNAL pr_state, nx_state: states;
SIGNAL temp: <data_type>;
---------- Phan mach to hop: -------------------------
PROCESS (pr_state)
BEGIN
      CASE pr_state IS

            WHEN state0 =>
                  temp <= <value>;
                  IF (condition) THEN nx_state <= state1;
                  ...
                  END IF;
            WHEN state1 =>
                  temp <= <value>;
                  IF (condition) THEN nx_state <= state2;
                  ...
                  END IF;
            WHEN state2 =>
                  temp <= <value>;
                  IF (condition) THEN nx_state <= state3;
                  ...
                  END IF;
            ...
      END CASE;
END PROCESS;
END <arch_name>;
```

So sánh khuôn mẫu của thiết kế kiểu 2 với thiết kế kiểu 1, chúng ta thấy chỉ có một sự khác nhau duy nhất, đó là xuất hiện tín hiệu trung gian temp. Tín hiệu này sẽ có tác dụng lưu trữ đầu ra của máy. Chỉ cho các giá trị chuyển thành đầu ra khi khi có sự thay đổi sự kiện clock.

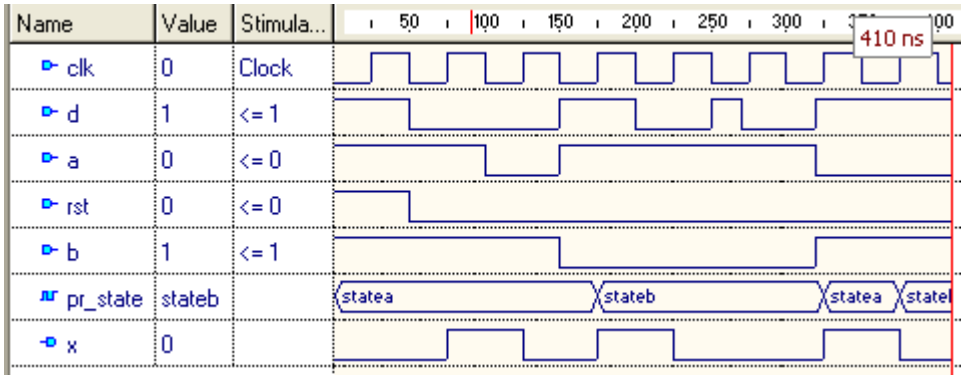
Ví dụ 8.3:

Chúng ta sẽ nhìn lại thiết kế của ví dụ 8.2. Tuy nhiên ở đây chúng ta muốn đầu ra là đồng bộ (chỉ thay đổi khi có sự kiện thay đổi clock). Vì vậy trong ví dụ này chúng ta sẽ thiết kế theo kiểu 2.

```
-------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
-------------------------------------------
ENTITY VD_FSM2 IS
      PORT ( a, b, d, clk, rst: IN BIT;
            x: OUT BIT);
END VD_FSM2;
-------------------------------------------
ARCHITECTURE VD_FSM2 OF VD_FSM2 IS
TYPE state IS (stateA, stateB);
SIGNAL pr_state, nx_state: state;
SIGNAL temp: BIT;
BEGIN
----- Phan mach day: ----------------------
PROCESS (rst, clk)
BEGIN
      IF (rst='1') THEN
            pr_state <= stateA;
      ELSIF (clk'EVENT AND clk='1') THEN
```

```
            x <= temp;
            pr_state <= nx_state;
      END IF;
END PROCESS;
---------- Phan mach to hop: -----------------
PROCESS (a, b, d, pr_state)
BEGIN
      CASE pr_state IS
            WHEN stateA =>
                  temp <= a;
                  IF (d='1') THEN nx_state <= stateB;
                  ELSE nx_state <= stateA;
                  END IF;
            WHEN stateB =>
                  temp <= b;
                  IF (d='1') THEN nx_state <= stateA;
                  ELSE nx_state <= stateB;
                  END IF;
      END CASE;
END PROCESS;
END VD_FSM2;
-------------------------------------------------
```

Ở đây chúng ta thấy có 2 flip – flop được sử dụng, một cái để mã hoá trạng thái của máy, một cái để lưu trữ đầu ra.
Bộ mô phỏng kết quả được chỉ ra trong hình dưới đây:



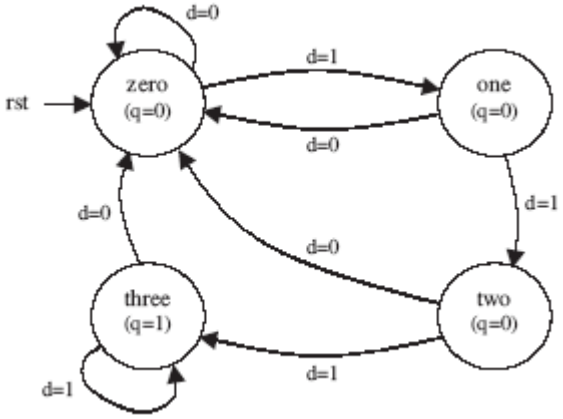
Hình 8.7.Kết quả mô phỏng cho ví dụ 8.3

Ví dụ 8.4. Bộ phát hiện chuỗi

Chúng ta muốn thiết kế một mạch mà đầu vào là luồng bit nối tiếp và đầu ra là 1 khi đầu có xuất hiện chuỗi “111”, là 0 trong các trường hợp còn lại

Đồ hình trạng thái của máy được chỉ ra trong hình 8. Ở đây chúng ta có 4 trạng thái và chúng ta quy ước là trạng thái zero, one, tow, three.

+ Trang thái 0 là trạng thái chờ 1 đầu tiên.
+ Trang thai 1 là trang thái đã có 1 và chờ 1 thứ 2
+ Trạng thái 2 là trạng thái đã có 11 và đang chờ 1 thứ 3.
+ Trạng thái 3 là trạng thái thu được xâu 111.



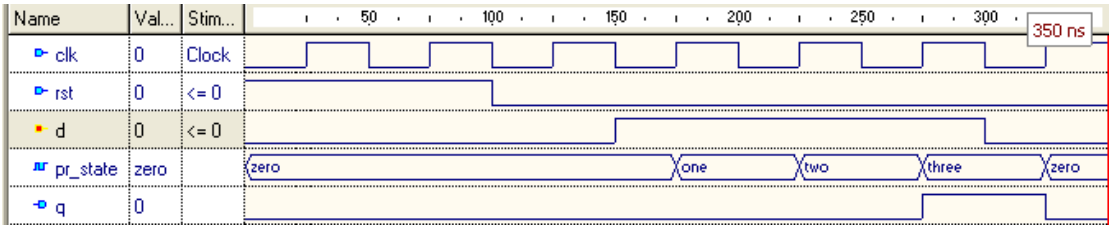
Hình 8.8. Sơ đồ trạng thái của bộ phát hiện chuỗi

Mã của máy được thiết kế như sau:

```
-----------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
-----------------------------------------------
ENTITY Bo_doan_xau IS
      PORT ( d, clk, rst: IN BIT;
               q: OUT BIT);
END Bo_doan_xau;
-------------------------------------------
ARCHITECTURE state_machine OF Bo_doan_xau IS
TYPE state IS (zero, one, two, three);
SIGNAL pr_state, nx_state: state;
BEGIN
--------- Phan mach day: --------------------
PROCESS (rst, clk)
BEGIN
      IF (rst='1') THEN
            pr_state <= zero;
      ELSIF (clk'EVENT AND clk='1') THEN
            pr_state <= nx_state;
      END IF;
END PROCESS;
---------- Phan mach to hop: ---------------
PROCESS (d, pr_state)
BEGIN
      CASE pr_state IS
            WHEN zero =>
                  q <= '0';
                  IF (d='1') THEN nx_state <= one;
                  ELSE nx_state <= zero;
                  END IF;
            WHEN one =>
                  q <= '0';
                  IF (d='1') THEN nx_state <= two;
                  ELSE nx_state <= zero;
                  END IF;
            WHEN two =>
```

```
                    q <= '0';
                    IF (d='1') THEN nx_state <= three;
                    ELSE nx_state <= zero;
                    END IF;
               WHEN three =>
                    q <= '1';
                    IF (d='0') THEN nx_state <= zero;
                    ELSE nx_state <= three;
                    END IF;
      END CASE;
END PROCESS;
END state_machine;
-----------------------------------------
```

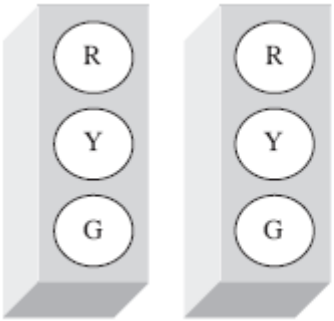
Kết quả mô phỏng sẽ như sau:



Hình 8.9.Kết quả mô phỏng cho bộ đoán nhận xâu.

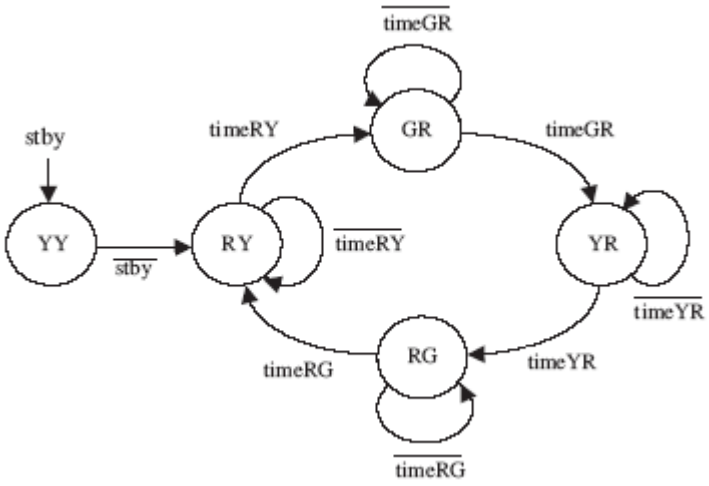
Ví dụ 8.5: Bộ điều khiển đèn giao thông (TLC)

Như đã giới thiệu ở phần mạch mạch tổ hợp, bộ điều khiển số là mạch ví dụ tốt để có thể thực hiện hiệu quả khi mô hình hoá máy trạng thái. Trong ví dụ này, chúng ta sẽ thiết kế một TLC với những đặc điểm được tóm lược như trong hình 8.10:



State	Operation Mode		
	REGULAR	TEST	STANDBY
	Time	Time	Time
RG	timeRG (30s)	timeTEST (1s)	---
RY	timeRY (5s)	timeTEST (1s)	---
GR	timeGR (45s)	timeTEST (1s)	---
YR	timeYR (5s)	timeTEST (1s)	---
YY	---	---	Indefinite

Hình 8.10.a. Sơ đồ nguyên lý hoạt động của TLC



Hình 8.10.b. Đồ hình trạng thái của TLC

Ở đây chúng ta thấy có 3 chế độ thao tác:

+ Chế độ bình thường: Ở chế độ này, mạch có 4 trạng thái, mỗi trạng thái là độc lập, thời gian lập trình ….?

+ Chế độ kiểm tra: Cho phép tất cả thời gian được lập trình trước được viết lên với 1 giá trị nhỏ, do vậy hệ thống có thể dễ dàng được kiểm tra trong suốt quá trình baỏ dưỡng.

+ Chế độ Standby: Nếu thiết lập hệ thống sẽ kích hoạt đèn vàng trong khi tín hiệu standby được kích hoạt.

Đồng thời 1 đông hồ tần số 60 HZ luôn hoạt động.

Mã thiết kế:

```
-------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
ENTITY Bodk_den_giao_thong IS
PORT ( clk, stby, test: IN STD_LOGIC;
r1, r2, y1, y2, g1, g2: OUT STD_LOGIC);
END Bodk_den_giao_thong;
-------------------------------------------------
ARCHITECTURE state_machine_be OF Bodk_den_giao_thong IS
CONSTANT timeMAX : INTEGER := 2700;
CONSTANT timeRG : INTEGER := 1800;
CONSTANT timeRY : INTEGER := 300;
CONSTANT timeGR : INTEGER := 2700;
CONSTANT timeYR : INTEGER := 300;
CONSTANT timeTEST : INTEGER := 60;
TYPE state IS (RG, RY, GR, YR, YY);
SIGNAL pr_state, nx_state: state;
SIGNAL time : INTEGER RANGE 0 TO timeMAX;
BEGIN
-------------Phan mach day: ----
PROCESS (clk, stby)
VARIABLE count : INTEGER RANGE 0 TO timeMAX;
BEGIN
      IF (stby='1') THEN
            pr_state <= YY;
```

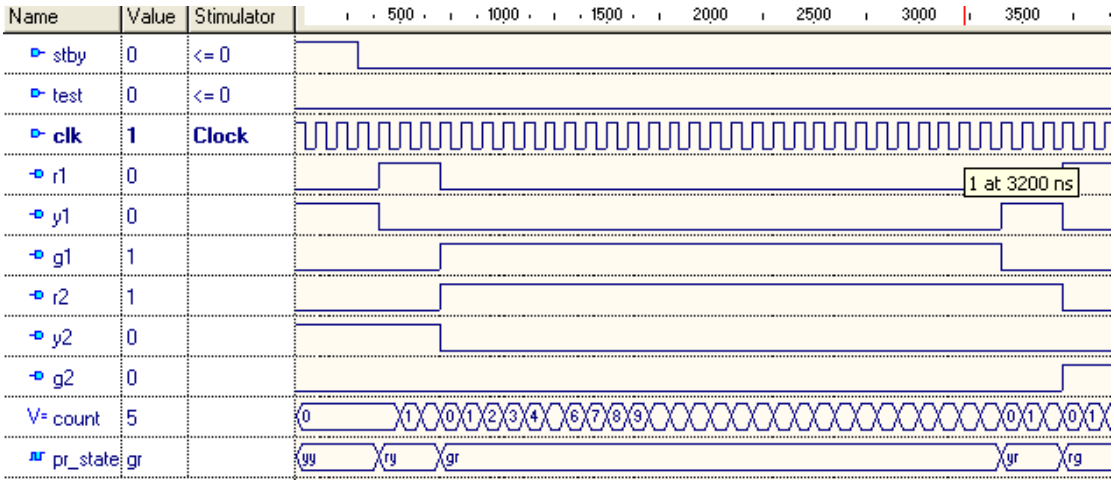
```
            count := 0;
      ELSIF (clk'EVENT AND clk='1') THEN
            count := count + 1;
            IF (count = time) THEN
            pr_state <= nx_state;
            count := 0;
            END IF;
      END IF;
END PROCESS;
----------- Phan mach to hop: ----
PROCESS (pr_state, test)
BEGIN
      CASE pr_state IS
            WHEN RG =>
                  r1<='1';r2<='0';y1<='0'; y2<='0'; g1<='0'; g2<='1';
                  nx_state <= RY;
                  IF (test='0') THEN time <= timeRG;
                  ELSE time <= timeTEST;
                  END IF;
            WHEN RY =>
                  r1<='1';r2<='0';y1<='0';y2<='1';g1<='0'; g2<='0';
                  nx_state <= GR;
                  IF (test='0') THEN time <= timeRY;
                  ELSE time <= timeTEST;
                  END IF;
            WHEN GR =>
                  r1<='0';r2<='1';y1<='0';y2<='0';g1<='1'; g2<='0';
                  nx_state <= YR;
                  IF (test='0') THEN time <= timeGR;
                  ELSE time <= timeTEST;
                  END IF;
            WHEN YR =>
                  r1<='0';r2<='1';y1<='1'; y2<='0'; g1<='0'; g2<='0';
                  nx_state <= RG;
                  IF (test='0') THEN time <= timeYR;
                  ELSE time <= timeTEST;
                  END IF;
            WHEN YY =>
                  r1<='0';r2<='0';y1<='1'; y2<='1'; g1<='0'; g2<='0';
                  nx_state <= RY;
      END CASE;
END PROCESS;
END state_machine_be;
-------------------------------------------------
```

Như ta thấy, số lượng Flip-flop đã dùng để thực hiện mạch là 15 cái: 3 cái cho lưu trữ trạng thái hiện tại, 12 cái còn lại cho bộ đếm.

Để có thể dễ dàng thấy kết quả mô phỏng, ở đây ta thực hiện giảm thời gian thực tế đi 100 lần.

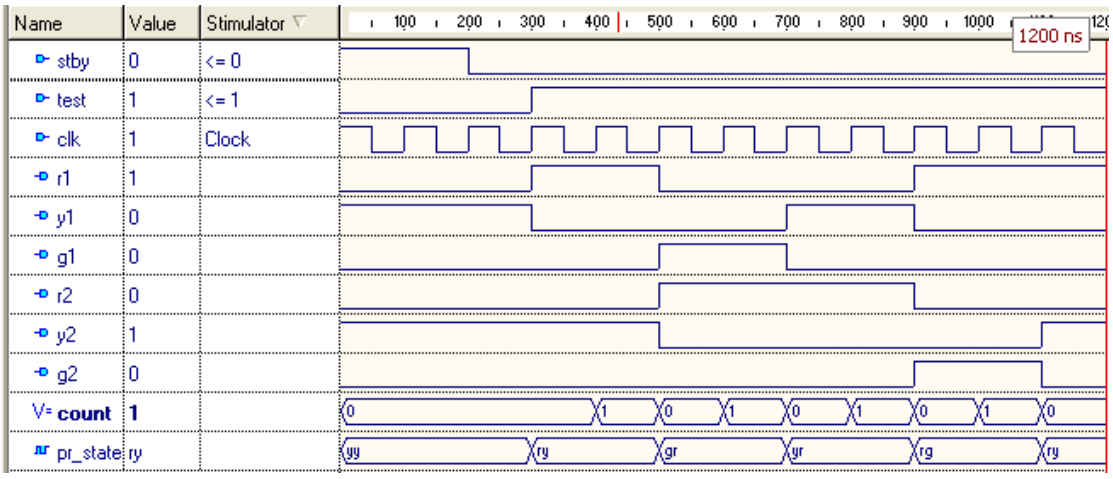
Kết quả mô phỏng được chỉ ra trong hình dưới đây:

+ Ở chế độ hoạt động bình thường (stby = 0, test = 0):



Hình 8.11.a. Kết quả mô phỏng TLC ở chế độ hd bình thường

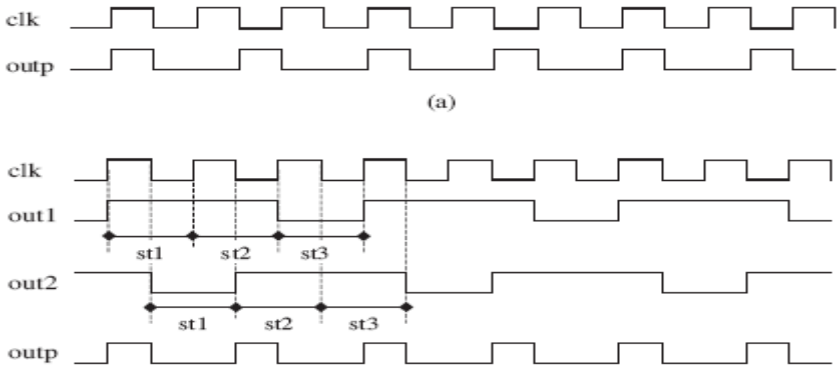
\+ Ở chế độ kiểm tra:



Hình 8.11.b. Kết quả mô phỏng TLC ở chế độ kiểm tra

Ví dụ 8.6: Bộ phát tín hiệu:

Chúng ta muốn thiết kế một mạch mà từ tín hiệu clock clk đưa ra tín hiệu như trong hình dưới đây:



Hình 8.12.Dạng tín hiệu cần tạo.

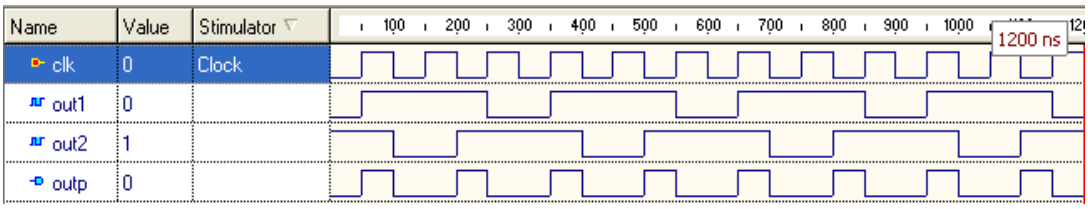
Ở đây mạch phải hoạt động ở cả 2 sườn của tín hiệu clk.

Mã chương trình:

```
---------------------------------------
ENTITY Bo_phat_tin_hieu IS
      PORT ( clk: IN BIT;
            outp: OUT BIT);
END Bo_phat_tin_hieu;
---------------------------------------
ARCHITECTURE state_machine OF Bo_phat_tin_hieu IS
TYPE state IS (one, two, three);
SIGNAL pr_state1, nx_state1: state;
SIGNAL pr_state2, nx_state2: state;
SIGNAL out1, out2: BIT;
BEGIN
----- Phan mach day cua may 1: ---
PROCESS(clk)
BEGIN
      IF (clk'EVENT AND clk='1') THEN
      pr_state1 <= nx_state1;
      END IF;
END PROCESS;
----- Phan mach day cua may 2: ---
PROCESS(clk)
BEGIN
      IF (clk'EVENT AND clk='0') THEN
            pr_state2 <= nx_state2;
      END IF;
END PROCESS;
---- Phan mach to hop cua may 1: -----
PROCESS (pr_state1)
BEGIN
      CASE pr_state1 IS
            WHEN one =>
                  out1 <= '0';
                  nx_state1 <= two;
            WHEN two =>
                  out1 <= '1';
                  nx_state1 <= three;
            WHEN three =>
                  out1 <= '1';
                  nx_state1 <= one;
      END CASE;
END PROCESS;
---- Phan mac«pt hop cua may 2: -----
PROCESS (pr_state2)
BEGIN
      CASE pr_state2 IS
            WHEN one =>
                  out2 <= '1';
                  nx_state2 <= two;
            WHEN two =>
                  out2 <= '0';
                  nx_state2 <= three;
            WHEN three =>
```

```
                out2 <= '1';
                nx_state2 <= one;
      END CASE;
END PROCESS;
      outp <= out1 AND out2;
END state_machine;
---------------------------------------
```

Kết quả mô phỏng:



Hình 8.13.Kết quả mô phỏng cho ví dụ 8.6

8.4. Kiểu mã hoá: từ nhị phân sang Onehot.

Để mã hoá trạng thái của máy trạng thái, chúng ta có thể chọn một trong vài kiểu có sẵn. Kiểu mã hoá mặc định là nhị phân. Ưu điểm của kiểu mã hoá này là nó yêu cầu số lượng flip-flop ít nhất. Trong trường hợp này, với n mạch flip-flop thì có thể chúng ta có thể mã hoá được 2^n trạng thái. Nhược điểm của kiểu mã hoá này là nó yêu cầu về logic nhiều hơn và nó chậm hơn so với những kiểu khác.

Cái cuối cùng là kiểu mã hoá onehot, với kiểu mã hoá này, chúng ta cần sử dụng 1 flip-flop cho 1 trạng thái. Vì vậy, nó đòi hỏi số lượng flip-flop lớn nhất. Trong trường hợp này, với n flip-flop (n bit) chỉ có thể mã hoá được n trạng thái. Nhưng bù lại, phương pháp này lại yêu cầu tính toán logic it nhất, và tốc độ nhanh nhất

Một kiểu nằm giữa 2 kiểu trên là kiểu mã hoá twohot (trong một trạng thái chỉ có 2 bit 1). Vì vậy với n flip-flop (n bit), thì chúng ta có thể mã hoá được n(n-1)/2 trạng thái.

Kiểu mã hoá onehot được giới thiệu trong các ứng dụng mà số lượng các flip-flop nhiều như trong các chip FPGA. Nhưng trong các mạch ASIC thì mã nhị phân lại được ưu tiên hơn.

Ví dụ: Giả sử chúng ta có một máy trạng thái có 8 trang thái như trong bảng dưới đây:

	Encoding Style		
STATE	BINARY	TWOHOT	ONEHOT
state0	000	00011	00000001
state1	001	00101	00000010
state2	010	01001	00000100
state3	011	10001	00001000
state4	100	00110	00010000
state5	101	01010	00100000
state6	110	10010	01000000
state7	111	01100	10000000

Bảng 8.1.Mã hoá trạng thái cho máy FSM 8 trạng thái

Với 8 trạng thái của máy này thì số lượng flip-flop được yêu cầu ứng với các kiểu mã hoá sẽ bằng:

+ 3 (=$\log_2 8$), ứng với kiểu mã hoá nhị phân.
+ 5 (n(n-1)/2= 8 => n = 5), ứng với kiểu mã hoá twohot
+ 8, ứng với kiểu mã hoá onehot.

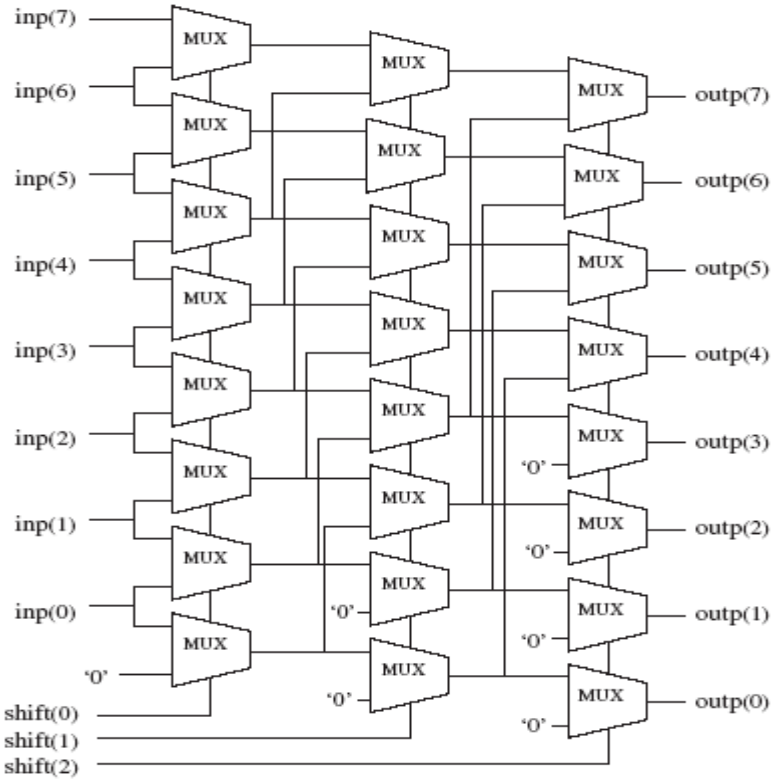
Chương 9: Thiết kế thêm các mạch

Phần này chúng ta sẽ trình bày các mạch sau:

+ Barrel shifter
+ Bộ so sánh không dấu và có dấu.
+ Bộ cộng
+ Bộ chia dấu chấm tĩnh.
+ Bộ điều khiển máy bán hàng.
+ Bộ nhận dữ liệu nối tiếp.
+ Bộ chuyển đổi song song sang nối tiếp.
+ SSD
+ Bộ phát tín hiệu
+ Bộ nhớ

9.1. Barrel Shifter.

Sơ đồ của mạch của bộ dịch barrel được chỉ ra trong hình 9.1. Đầu vào là vector 8 bit. Đầu ra là phiên bản dịch của đầu vào, với lượng dịch được định nghĩa bởi 8 đầu vào "shift" (từ o đến 7). Mạch gồm có 3 bộ dịch barrel riêng lẻ, mỗi một cái giống như trong ví dụ 6.9. Nhưng chúng ta phải chu ý rằng, barrel đầu tiên có chỉ có 1 đầu "0" được kết nối với một bộ dồn kênh, trong khi barrel thứ 2 có 2 đầu vào "0" và barrel cuối cùng có tới 4 đầu vào "0". Để vector lớn hơn thì chúng ta phải dữ 2 đầu vào là "0". Ví dụ nếu shift = "001" thì chỉ barrel đầu tiên gây ra dịch, còn nếu shift = "111" thì tất các đều gây ra dịch.



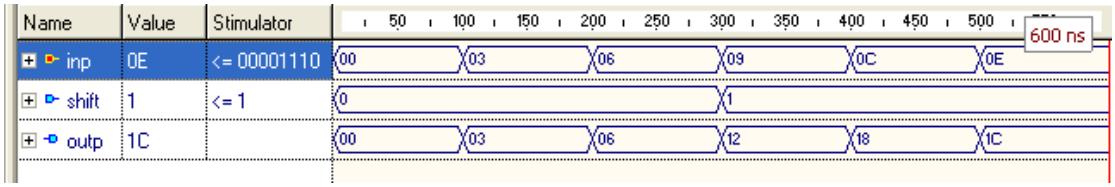
Hình 9.1. Bộ dịch barrel

Mã thiết kế sẽ như sau:

```
-----------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------
ENTITY barrel IS
      PORT ( inp: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
            shift: IN STD_LOGIC_VECTOR (2 DOWNTO 0);
            outp: OUT STD_LOGIC_VECTOR (7 DOWNTO 0));
END barrel;
-----------------------------------------
ARCHITECTURE behavior OF barrel IS
BEGIN
PROCESS (inp, shift)
VARIABLE temp1: STD_LOGIC_VECTOR (7 DOWNTO 0);
VARIABLE temp2: STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
---- Bo dich thu nhat -----
      IF (shift(0)='0') THEN
            temp1 := inp;
      ELSE
            temp1(0) := '0';
            FOR i IN 1 TO inp'HIGH LOOP
                  temp1(i) := inp(i-1);
            END LOOP;
      END IF;
---- Bo dich thu 2 -----
```

```
      IF (shift(1)='0') THEN
            temp2 := temp1;
      ELSE
            FOR i IN 0 TO 1 LOOP
                  temp2(i) := '0';
            END LOOP;
            FOR i IN 2 TO inp'HIGH LOOP
                  temp2(i) := temp1(i-2);
            END LOOP;
      END IF;
---- Bo dich thu 3 -----
      IF (shift(2)='0') THEN
            outp <= temp2;
      ELSE
            FOR i IN 0 TO 3 LOOP
                  outp(i) <= '0';
            END LOOP;
            FOR i IN 4 TO inp'HIGH LOOP
                  outp(i) <= temp2(i-4);
            END LOOP;
      END IF;
END PROCESS;
END behavior;
-------------------------------------------
```

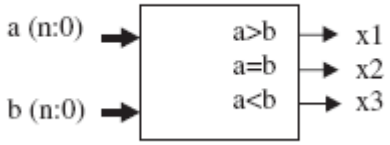
Kết quả mô phỏng:



Hình 9.2.Kết quả mô phỏng cho bộ dịch barrel

9.2. Bộ so sánh không dấu và có dấu.

Hình 9.3 hiện lên sơ đồ của bộ so sánh. Kích thước của vector được so sánh là generic (n+1). 3 đầu ra phải được cung cấp là: 1 đầu ra là a>b, 1 đầu ra là a = b, đầu ra còn lại là a < b. 3 giải pháp được giới thiệu : đầu tiên xét a và b là các số có dấu, trong khi 2 giải pháp còn lại là các số không dấu. Kết quả mô phỏng sẽ cho chúng ta thấy rõ hơn.



Hình 9.3.Mô hình của bộ so sánh

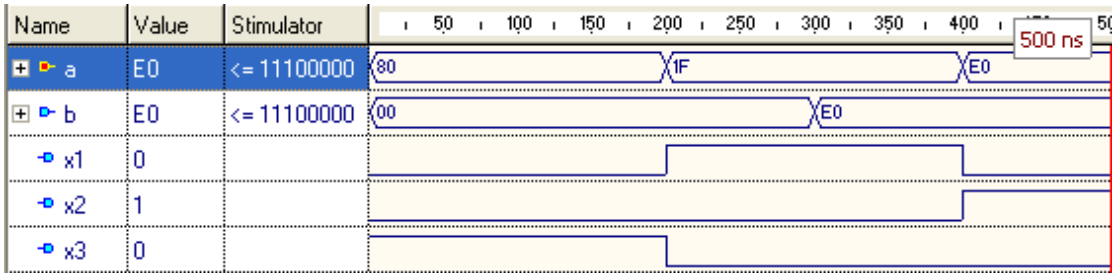
Bộ so sánh có dấu:

Để làm việc với số có dấu hoặc số không dấu thì chúng ta đều phải khai báo gói std_logic_arith (cụ thể chúng ta sẽ thấy trong đoạn mã dưới đây).

Mã thiết kế bộ so sánh có dấu:

```
---- Bo so sanh co dau: ----------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; -- can thiet!
---------------------------------------
ENTITY Bo_so_sanh_co_dau IS
GENERIC (n: INTEGER := 7);
PORT (a, b: IN SIGNED (n DOWNTO 0);
           x1, x2, x3: OUT STD_LOGIC);
END Bo_so_sanh_co_dau;
---------------------------------------
ARCHITECTURE arc OF Bo_so_sanh_co_dau IS
BEGIN
      x1 <= '1' WHEN a > b ELSE '0';
      x2 <= '1' WHEN a = b ELSE '0';
      x3 <= '1' WHEN a < b ELSE '0';
END arc;
---------------------------------------
```

Kết quả mô phỏng:



Hình 9.4. Kết quả mô phỏng bộ so sánh có dấu

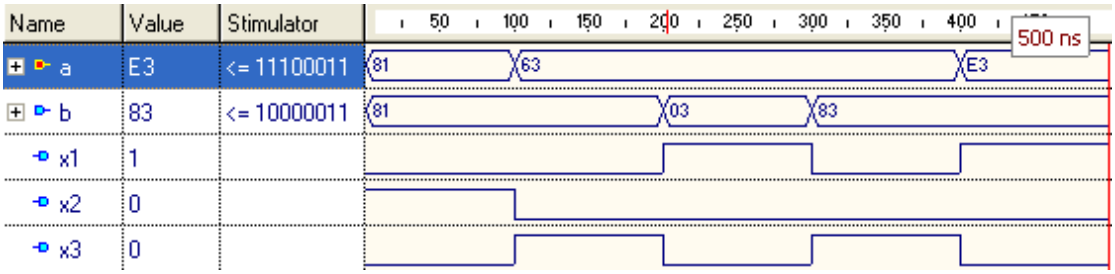
Bộ so sánh không dấu 1:

Phần mã VHDL sau đây là bản sao của phần mã đã được trình bày (ở bộ so sánh không dấu).

```
---- Bo so sanh khong dau 1: -----------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all; --rat can thiet!
---------------------------------------
```

```
ENTITY Bo_so_sanh_khong_dau1 IS
GENERIC (n: INTEGER := 7);
PORT (a, b: IN UNSIGNED (n DOWNTO 0);
          x1, x2, x3: OUT STD_LOGIC);
END Bo_so_sanh_khong_dau1;
---------------------------------------
ARCHITECTURE arc OF Bo_so_sanh_khong_dau1 IS
BEGIN
      x1 <= '1' WHEN a > b ELSE '0';
      x2 <= '1' WHEN a = b ELSE '0';
      x3 <= '1' WHEN a < b ELSE '0';
END arc;
---------------------------------------
```

Kết quả:



Hình 9.5.1.Kết quả bộ so sánh không dấu 1

Bộ so sánh không dấu 2:

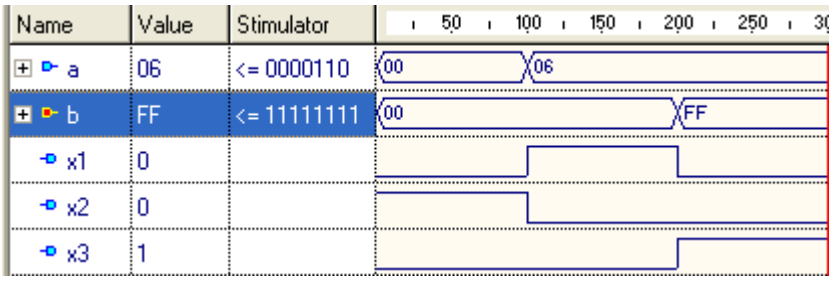
Bộ so sánh không dấu có thể cũng được thực hiện với STD_LOGIC_VECTORS, trong trường hợp này không cần thiết phải khai báo std_logic_arith.

Mã thiết kế sẽ như sau:

---- Bo so sanh khong dau: ----------

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY comparator IS
GENERIC (n: INTEGER := 7);
PORT (a, b: IN STD_LOGIC_VECTOR (n DOWNTO 0);
          x1, x2, x3: OUT STD_LOGIC);
END comparator;
---------------------------------------
ARCHITECTURE unsigned OF comparator IS
BEGIN
          x1 <= '1' WHEN a > b ELSE '0';
          x2 <= '1' WHEN a = b ELSE '0';
          x3 <= '1' WHEN a < b ELSE '0';
 END unsigned;
```

Mô phỏng kết quả:



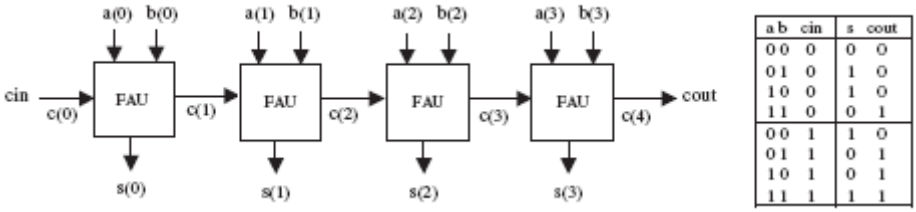
Hình 9.5.2. Kết quả của bộ so sánh không dấu2

9.3. Bộ cộng Carry Ripple và bộ cộng Carry Look Ahead.

Carry ripple và carry look ahead là 2 phương pháp cổ điển để thiết kế cácc bộ cộng. Phương pháp đầu tiên có thuận lợi là yêu cầu phần cứng ít, trong khi cái thứ hai lại nhanh hơn.

+ Bộ cộng carry ripple:

Hình9.6 chỉ ra 1 bộ cộng ripple cary 4 bit không dấu:



Hình 9.6. Sơ đồ bộ cộng ripple carry

Trên sơ đồ ta có thể thấy, với mỗi bit, một đơn vị bộ cộng đầy đủ sẽ được thực hiện. Bảng thật của bộ cộng đầy đủ được chỉ ra bên cạnh sơ đồ, trong đó a, b là các bít đầu vào, cin là bit nhớ vào, s là bit tổng, cout là bit nhớ ra. Từ bảng thật ta dễ dàng tính được:

s = a xor b xor cin

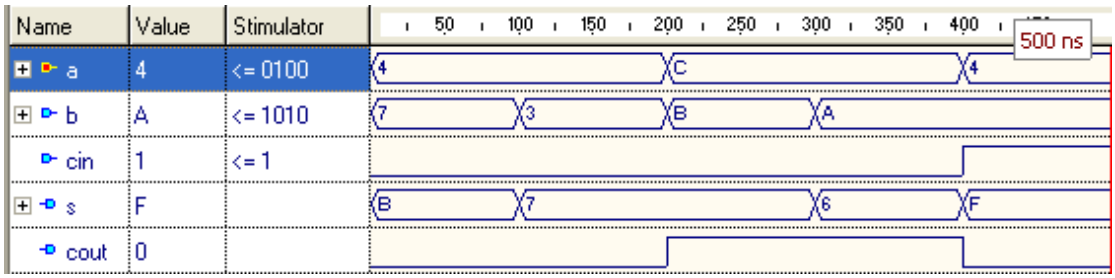
cout = (a and b) xor (a and cin) xor (b xor cin)

Từ công thức trên ta xây dựng chương trình VHDL như sau (Ở đây chúng ta có thể áp dụng cho bất kỳ số lượng đầu vào nào):

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY Bo_cong_carry_ripple IS
GENERIC (n: INTEGER := 4);
PORT ( a, b: IN STD_LOGIC_VECTOR (n-1 DOWNTO 0);
           cin: IN STD_LOGIC;
           s: OUT STD_LOGIC_VECTOR (n-1 DOWNTO 0);
           cout: OUT STD_LOGIC);
END Bo_cong_carry_ripple;
-------------------------------------------
```

```
ARCHITECTURE arc OF Bo_cong_carry_ripple IS
SIGNAL c: STD_LOGIC_VECTOR (n DOWNTO 0);
BEGIN
      c(0) <= cin;
      G1: FOR i IN 0 TO n-1 GENERATE
                  s(i) <= a(i) XOR b(i) XOR c(i);
                  c(i+1) <= (a(i) AND b(i)) OR
                                    (a(i) AND c(i)) OR
                                    (b(i) AND c(i));
            END GENERATE;
      cout <= c(n);
END arc;
-------------------------------------------
```

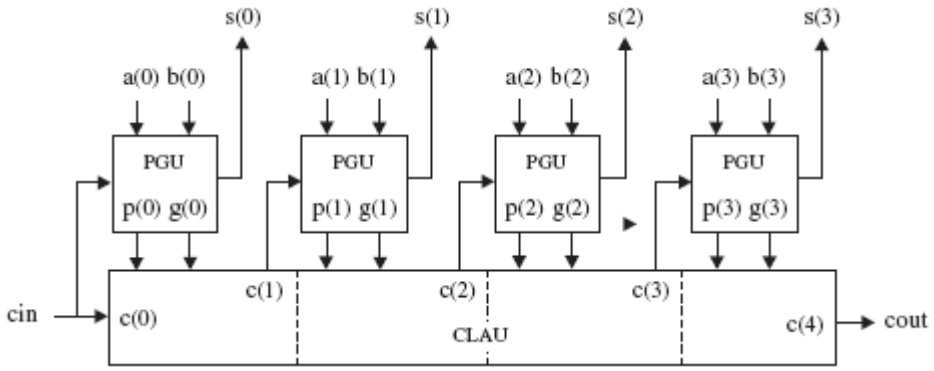
Kết quả mô phỏng:



Hình 9.7. Kết quả mô phỏng cho bộ cộng ripple carry

+ Bộ cộng carry look ahead:

Sơ đồ bộ cộng carry look ahead 4 bit được chỉ ra trong hình 9.8.1 dưới đây:



Hình 9.8.1. Sơ đồ bộ cộng carry look ahead

Mạch được hoạt động dựa trên các khái niệm generate và propagate. Chính đặc điểm này đã làm cho bộ cộng này thực hiện với tốc độ nhanh hơn so với bộ cộng trước.

Giả sử 2 đầu vào là 2 bit a,b thì 2 tín hiệu p(propagate) và g(generate) được tính như sau:

g = a and b

p = a or b

Nếu chúng ta xem a, b là các vector:

a = a(n-1)…a(1)a(0) ; b = b(n-1)…b(1)b(0)

thì g, p được tính như sau:

p = p(n-1)…p(1)p(0); g = g(n-1)…g(1)g(0)

Trong đó:

g(i) = a(i) and b(i)

p(i) = a(i) or b(i)

Lúc này vector nhớ sẽ là: c = c(n-1)…c(1)c(0), trong đó:

c(0) = cin

c(1) = c(0)p(0) + g(0)

c(2) = c(0)p(0)p(1) + g(0)p(1) + g(1)

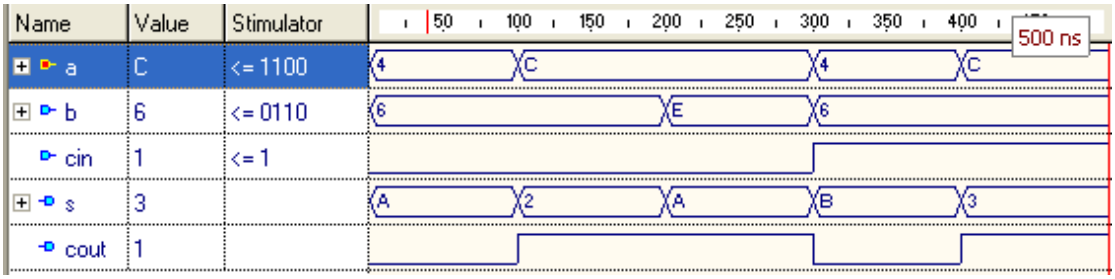
c(i) = c(i-1)p(i-1) + g(i-1)

Từ công thức tình trên, chúng ta viết chương trình thiết kế bộ cộng carry look ahead 4 bit như sau:

```
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY Bo_cong_carry_look_ahead IS
PORT ( a, b: IN STD_LOGIC_VECTOR (3 DOWNTO 0);
           cin: IN STD_LOGIC;
           s: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
           cout: OUT STD_LOGIC);
END Bo_cong_carry_look_ahead;
-------------------------------------------
ARCHITECTURE            Bo_cong_carry_look_ahead            OF
Bo_cong_carry_look_ahead IS
SIGNAL c: STD_LOGIC_VECTOR (4 DOWNTO 0);
SIGNAL p: STD_LOGIC_VECTOR (3 DOWNTO 0);
SIGNAL g: STD_LOGIC_VECTOR (3 DOWNTO 0);
BEGIN
---- PGU: --------------------------------
     G1: FOR i IN 0 TO 3 GENERATE
           p(i) <= a(i) XOR b(i);
           g(i) <= a(i) AND b(i);
           s(i) <= p(i) XOR c(i);
     END GENERATE;
---- CLAU: -------------------------------
     c(0) <= cin;
     c(1) <= (cin AND p(0)) OR
     g(0);
     c(2) <= (cin AND p(0) AND p(1)) OR
     (g(0) AND p(1)) OR
     g(1);
     c(3) <= (cin AND p(0) AND p(1) AND p(2)) OR
                     (g(0) AND p(1) AND p(2)) OR
                     (g(1) AND p(2)) OR g(2);
     c(4) <= (cin AND p(0) AND p(1) AND p(2) AND p(3)) OR
```

```
                    (g(0) AND p(1) AND p(2) AND p(3)) OR
                    (g(1) AND p(2) AND p(3)) OR
                    (g(2) AND p(3)) OR g(3);
      cout <= c(4);
END Bo_cong_carry_look_ahead;
-----------------------------------------
```

Kết quả mô phỏng:



Hình 9.8.2. Kết quả mô phỏng cho bộ cộng carry look ahead

9.4. Bộ chia dấu chấm tĩnh.

Trước khi đi vào thiết kế, chúng ta cần phải nhắc lại thuật toán chia:

Thuật toán chia:

Mục đích của thuật toán là chúng ta cần tính y = a/b trong đó a, b là những số cùng có (n+1) bit.

Thuật toán được thể hiện trong bảng 9.9, trong đó a = “1011” (= $(11)_{10}$) và b = “0011” (=$(3)_{10}$). Kết quả sẽ thu được: thương y = “0011” (=$(3)_{10}$) và số dư r = “0010” (=$(2)_{10}$).

Chỉ số	Đầu vào a	So sánh	Đầu vào b	y	Thao tác cho cột a
3	1011	<	0011000	0	Không làm gì
2	1011	<	0001100	0	Không làm gì
1	1011	>	0000110	1	Trừ cột a cho cột b
0	0101	>	0000011	1	Trừ cột a cho cột b
	0010				

Hình 9.9. Thuật toán chia

Giải thích thuật toán:

+ Đầu tiên chuyển số chia thành số 2n+1 bit bằng cách thêm vào sau n -1 bit 0 , số bị chia vẫn giữ nguyên.
+ So sánh số bị chia với số chia . Nếu số bị chia lớn hơn hoặc bằng số chia thì gán y =1 và thay số bị chia bằng hiệu của số bị chia với số chia. Ngược lại thì y =0
+ Quá trình thực hiện liên tục cho đến khi hết n lần.
+ Thương là dãy bit của y, số dư là sô bị chia cuối cùng.

Để thiết kế bộ chia này thì chúng ta có 2 phương pháp: Cả 2 phương pháp đều thực hiện theo mã tuần tự: Phương pháp thứ nhất chỉ thực hiện bằng câu lện if, phương pháp thứ 2 thực hiện bằng cả câu lện if và loop.

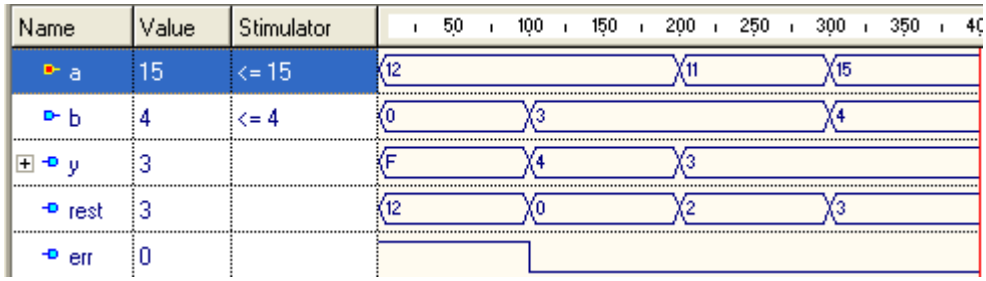
Mã thiết kế bộ chia sẽ như sau:

Thiết kế theo phương pháp 1:

```
----- Phuong phap 1: step-by-step -------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY Bo_chia IS
PORT ( a, b: IN INTEGER RANGE 0 TO 15;
           y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
           rest: OUT INTEGER RANGE 0 TO 15;
           err : OUT STD_LOGIC);
END Bo_chia;
-------------------------------------------------
ARCHITECTURE arc OF Bo_chia IS
BEGIN
PROCESS (a, b)
VARIABLE temp1: INTEGER RANGE 0 TO 15;
VARIABLE temp2: INTEGER RANGE 0 TO 15;
BEGIN
----- Khoi tao vaµ bat loi: -------
      temp1 := a;
      temp2 := b;
      IF (b=0) THEN err <= '1';
      ELSE err <= '0';
      END IF;
----- y(3): ---------------------------
      IF (temp1 >= temp2 * 8) THEN
            y(3) <= '1';
            temp1 := temp1 - temp2*8;
      ELSE y(3) <= '0';
```

```
      END IF;
----- y(2): ---------------------------
      IF (temp1 >= temp2 * 4) THEN
            y(2) <= '1';
            temp1 := temp1 - temp2 * 4;
      ELSE y(2) <= '0';
      END IF;
----- y(1): ---------------------------
      IF (temp1 >= temp2 * 2) THEN
            y(1) <= '1';
            temp1 := temp1 - temp2 * 2;
      ELSE y(1) <= '0';
      END IF;
----- y(0): ---------------------------
      IF (temp1 >= temp2) THEN
            y(0) <= '1';
            temp1 := temp1 - temp2;
      ELSE y(0) <= '0';
      END IF;
----- Phan du: ----------------------
      rest <= temp1;
      END PROCESS;
END arc;
-------------------------------------------------
```

Kết quả mô phỏng:



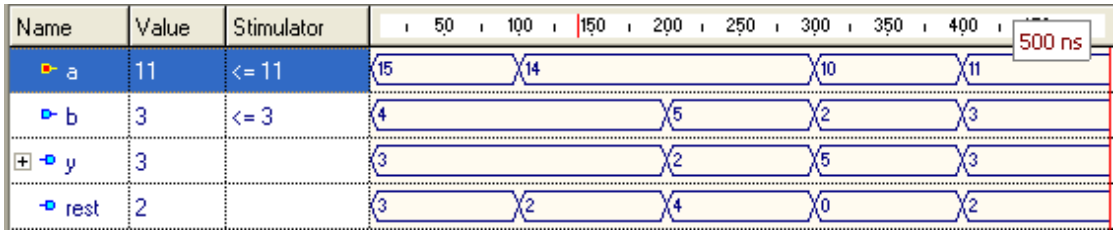
Hình 9.10.1. Kết quả mô phỏng bộ chia

Thiết kế theo phương pháp 2:

```
------ Phuong phap 2:----------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY Bo_chia2 IS
GENERIC(n: INTEGER := 3);
PORT ( a, b: IN INTEGER RANGE 0 TO 15;
            y: OUT STD_LOGIC_VECTOR (3 DOWNTO 0);
            rest: OUT INTEGER RANGE 0 TO 15;
            err : OUT STD_LOGIC);
END Bo_chia2;
-------------------------------------------------
ARCHITECTURE arc OF Bo_chia2 IS
BEGIN
```

```
PROCESS (a, b)
VARIABLE temp1: INTEGER RANGE 0 TO 15;
VARIABLE temp2: INTEGER RANGE 0 TO 15;
BEGIN
----- Khoi tao gia tri va bat loi: -------
      temp1 := a;
      temp2 := b;
      IF (b=0) THEN err <= '1';
      ELSE err <= '0';
      END IF;
----- thuong: ------------------------------
      FOR i IN n DOWNTO 0 LOOP
            IF(temp1 >= temp2 * 2**i) THEN
                  y(i) <= '1';
                  temp1 := temp1 - temp2 * 2**I;
            ELSE y(i) <= '0';
            END IF;
      END LOOP;
----- phan du: ----------------------
      rest <= temp1;
END PROCESS;
END arc;
-------------------------------------------------
```

Kết quả mô phỏng:



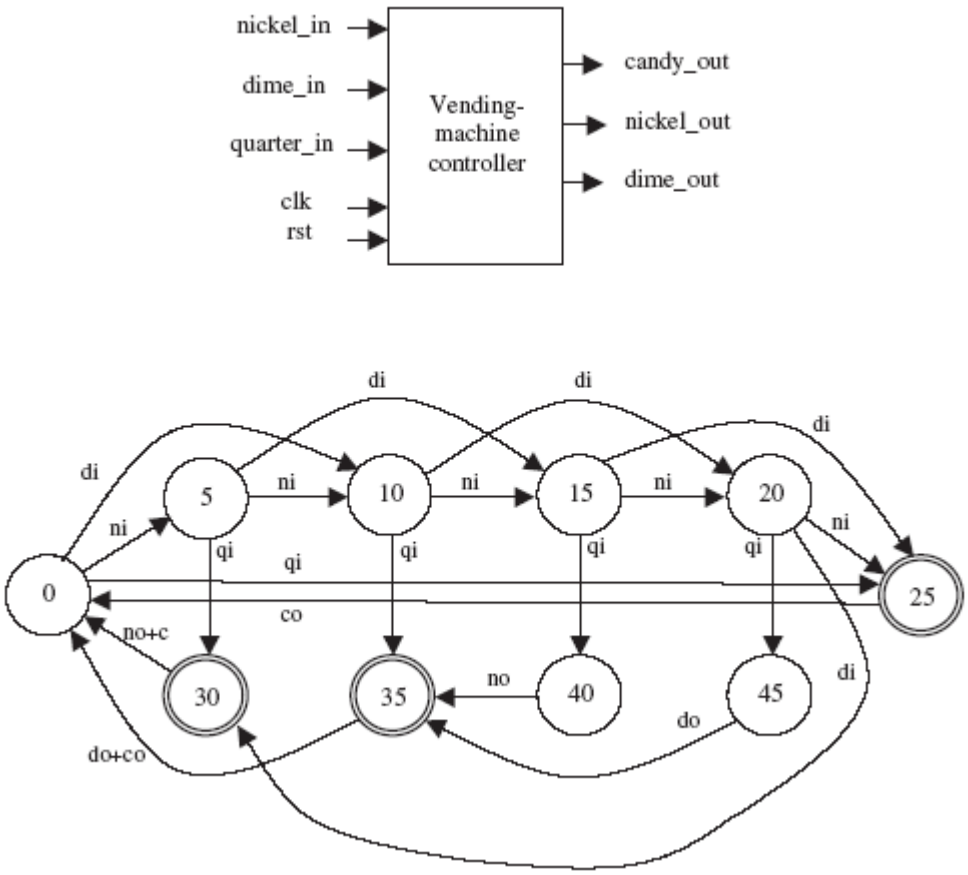
Hình 9.10.2.Kết quả mô phong bộ chia thứ 2

9.5. Bộ điều khiển máy bán hàng.

Trong ví dụ này, chúng ta sẽ thiết kế bộ điều khiển máy bán hàng, máy bán hàng sẽ bán các thanh kẹo với giá 25 xu. Chúng ta sẽ thiết kế theo mô hình máy FSM. Đầu ra và đầu vào của bộ điều khiển được thể hiện trong hình 9.11.

Tín hiệu vào là nickel_in, dime_in, và quarter_in thông báo rằng một đồng tiền tương ứng được gửi vào tài khoản. Ngoài ra còn có 2 đầu vào điều khiển: đầu vào reset (rst) và đầu vào clock (clk). Bộ điều khiển trả lời bằng 3 tín hiệu đầu ra: candy_out (để phân phát thanh kẹo), nickel_out và dime_out(cập nhật lại thay đổi).

Trên hình 9.11 cũng chỉ ra đồ hình trạng thái của máy FSM. Các số bên trong các vòng tròn biểu diễn tổng tài khoản của khách hàng (chỉ có các nickel, dime và quarter là được chấp nhận).



Hình 9.11. Đồ hình trạng thái của bộ điều khiển máy bán hàng

Trạng thái 0 là trạng thái là trạng thái không làm gì cả. Từ đó nếu 1 đồng nickel được gửi vào tài khoản, máy sẽ chuyển trạng thái đến trạng thái 5, nếu 1 đồng dime được gửi vào tài khoản thì máy chuyển tới trạng thái 10 hoặc nếu 1 đồng quarter thì máy sẽ chuyển đến trạng thái 25. Tình huống tương tự sẽ được lặp lại cho tất cả các trạng thái, cho tới trạng thai 20. Nếu trạng thái 25 được xác nhận, thì thanh kẹo được phân phát và không chuyển đổi. Tuy nhiên nếu trạng thái 40 được xác nhận thì a nickel được trả lại, bởi vậy trạng thái sẽ chuyển tới trạng thái 35, đó là 1 trạng thái mà 1 dime được trả lại và 1 candy bar được phân phát. Có 3 trạng thái tạo ra chu trình kép, đó là từ 1 thanh kẹo được phân phát và máy trở lại trạng thái 0. Bài toán này sẽ được chia thành 2 phần:

+ Trong phần đầu: diện mạo cơ bản liên quan đến thiết kế bộ điều khiển máy bán hàng (như trong hình 9.11) .

+ Trong phần 2: Các chức năng mở rộng được thêm vào.

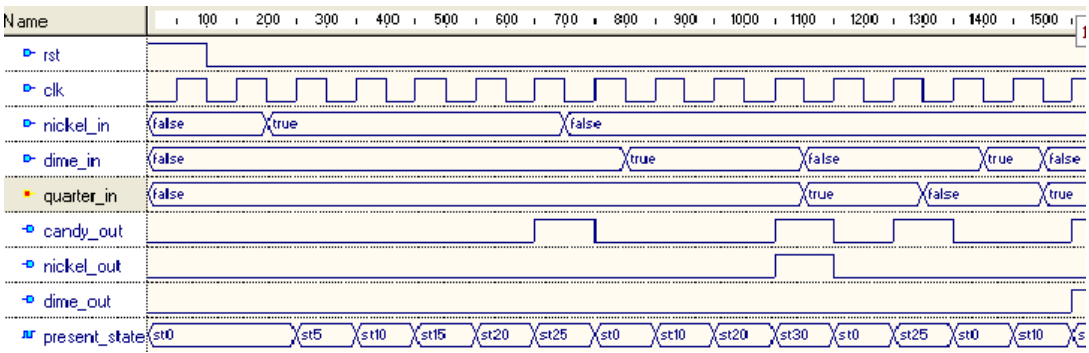
Ở đây chúng ta chỉ nghiên cứu phần một của bài toán: Nhìn vào đồ hình trạng thái của máy ở hình 9.11, chúng ta thấy có 10 trạng thái, như vậy cần có 4 bit để mã hoá các trạng thái, tức là cần sử dụng 4 flip-flop.

Mã thiết kế sẽ như sau:

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY Bo_dieu_khien_may_bh IS
PORT ( clk, rst: IN STD_LOGIC;
            nickel_in, dime_in, quarter_in: IN BOOLEAN;
            candy_out, nickel_out, dime_out: OUT STD_LOGIC);
END Bo_dieu_khien_may_bh;
-------------------------------------------------------
ARCHITECTURE state_machine OF Bo_dieu_khien_may_bh IS
TYPE state IS (st0, st5, st10, st15, st20, st25,
                        st30, st35, st40, st45);
SIGNAL present_state, next_state: STATE;
BEGIN
---- Lower section of the FSM (Sec. 8.2): ---------
PROCESS (rst, clk)
BEGIN
      IF (rst='1') THEN
            present_state <= st0;
      ELSIF (clk'EVENT AND clk='1') THEN
      present_state <= next_state;
      END IF;
END PROCESS;
---- Upper section of the FSM (Sec. 8.2): ---------
PROCESS (present_state, nickel_in, dime_in, quarter_in)
BEGIN
      CASE present_state IS
            WHEN st0 =>
                  candy_out <= '0';
                  nickel_out <= '0';
                  dime_out <= '0';
                  IF (nickel_in) THEN next_state <= st5;
                  ELSIF (dime_in) THEN next_state <= st10;
                  ELSIF (quarter_in) THEN next_state <= st25;
                  ELSE next_state <= st0;
                  END IF;
            WHEN st5 =>
                  candy_out <= '0';
                  nickel_out <= '0';
                  dime_out <= '0';
                  IF (nickel_in) THEN next_state <= st10;
                  ELSIF (dime_in) THEN next_state <= st15;
                  ELSIF (quarter_in) THEN next_state <= st30;
                  ELSE next_state <= st5;
                  END IF;
                  WHEN st10 =>
                  candy_out <= '0';
                  nickel_out <= '0';
                  dime_out <= '0';
                  IF (nickel_in) THEN next_state <= st15;
                  ELSIF (dime_in) THEN next_state <= st20;
                  ELSIF (quarter_in) THEN next_state <= st35;
                  ELSE next_state <= st10;
                  END IF;
            WHEN st15 =>
                  candy_out <= '0';
                  nickel_out <= '0';
                  dime_out <= '0';
```

```
                        IF (nickel_in) THEN next_state <= st20;
                        ELSIF (dime_in) THEN next_state <= st25;
                        ELSIF (quarter_in) THEN next_state <= st40;
                        ELSE next_state <= st15;
                        END IF;
                WHEN st20 =>
                        candy_out <= '0';
                        nickel_out <= '0';
                        dime_out <= '0';
                        IF (nickel_in) THEN next_state <= st25;
                        ELSIF (dime_in) THEN next_state <= st30;
                        ELSIF (quarter_in) THEN next_state <= st45;
                        ELSE next_state <= st20;
                        END IF;
                WHEN st25 =>
                        candy_out <= '1';
                        nickel_out <= '0';
                        dime_out <= '0';
                        next_state <= st0;
                WHEN st30 =>
                        candy_out <= '1';
                        nickel_out <= '1';
                        dime_out <= '0';
                        next_state <= st0;
                WHEN st35 =>
                        candy_out <= '1';
                        nickel_out <= '0';
                        dime_out <= '1';
                        next_state <= st0;
                WHEN st40 =>
                        candy_out <= '0';
                        nickel_out <= '1';
                        dime_out <= '0';
                        next_state <= st35;
                WHEN st45 =>
                        candy_out <= '0';
                        nickel_out <= '0';
                        dime_out <= '1';
                        next_state <= st35;
        END CASE;
END PROCESS;
END state_machine;
-------------------------------------------------------
```

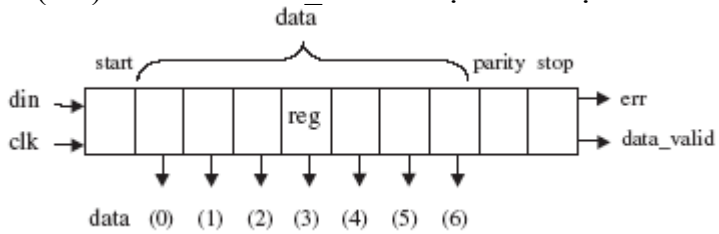
Kết quả mô phỏng:



Hình 9.12.Kết quả mô phỏng bộ điều khiển máy bán hàng

9.6. Bộ nhận dữ liệu nối tiếp.

Sơ đồ khối của bộ nhận dữ liệu nối tiếp được chỉ ra trong hình 9.13. Nó bao gồm một đầu vào dữ liệu nối tiếp (din) và một đầu ra dữ liệu song song (data(6:0)). Ngoài ra còn có tín hiệu điều khiển clk (tín hiệu clock). Hai tín hiệu giám sát được tạo ra bởi mạch là: tín hiệu err (error) và tín hiệu data_valid. Đầu vào xử lý chứa 10 bít. Bit đầu tiên là bit bắt đầu, nếu bit là 1 thì mạch bắt đầu nhận dữ liệu. 7 bit tiếp theo là các bit dữ liệu hoạt động. Bit thứ 9 là bit chẵn lẻ: bit này = '0' nếu số lượng các bit 1 trong dữ liệu la chẵn và bằng '1' trong trường hợ còn lại. Bit 10 là bit stop: bit này sẽ mang giá trị là 1 nếu quá trình chuyển đổi là đúng. Một lỗi được phát hiện khi bit chẵn lẻ không được kiểm tra hoặc bit stop không phải la '1'. Khi quá trình nhận kết thúc mà không có lỗi nào được phát hiện thì dữ liệu được lưu trữ trong các thanh ghi bên trong sẽ chuyển vào data(6:0) và đầu ra data_valid được xác nhận.



Hình 9.13. Sơ đồ bộ nhận dữ liệu nối tiếp

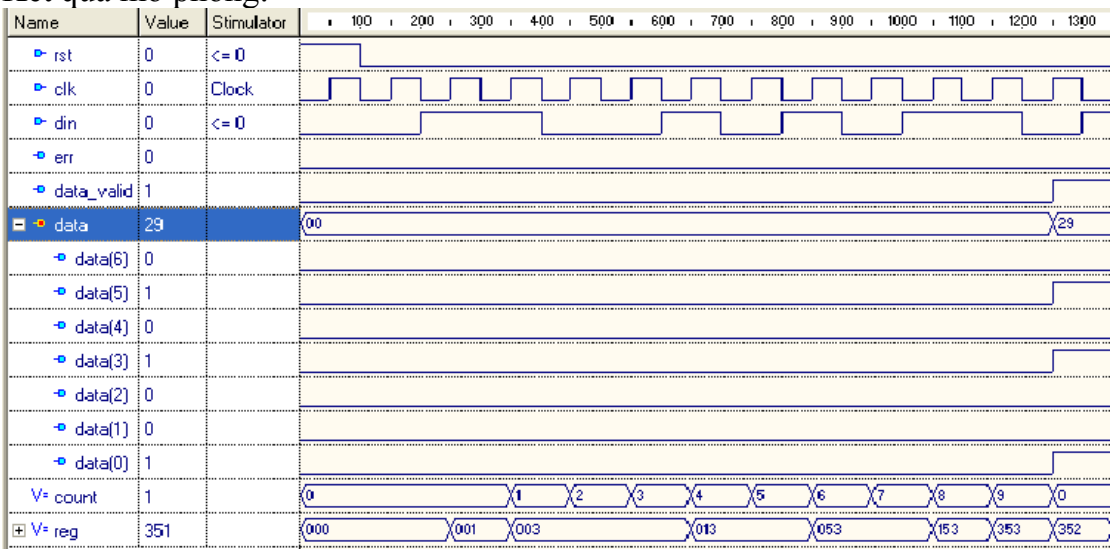
Để thiết kế mạch này chúng ta sẽ sử dụng một vài biến để làm các biến đếm, biến xác nhận số bit nhận được, biến lưu trữ dữ liệu, biến tính toán lỗi và biến trung gian.

Mã thiết kế bộ nhận dữ liệu nối tiếp sẽ như sau:

```
-------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------
ENTITY Bo_nhan_du_lieu_nt IS
PORT ( din, clk, rst: IN BIT;
          data: OUT BIT_VECTOR (6 DOWNTO 0);
          err, data_valid: OUT BIT);
END Bo_nhan_du_lieu_nt;
-------------------------------------------
ARCHITECTURE arc OF Bo_nhan_du_lieu_nt IS
BEGIN
PROCESS (rst, clk)
VARIABLE count: INTEGER RANGE 0 TO 10;
VARIABLE reg: BIT_VECTOR (10 DOWNTO 0);
VARIABLE temp : BIT;
BEGIN
     IF (rst='1') THEN
          count:=0;
          reg := (reg'RANGE => '0');
```

```
            temp := '0';
            err <= '0';
            data_valid <= '0';
      ELSIF (clk'EVENT AND clk='1') THEN
            IF (reg(0)='0' AND din='1') THEN
                  reg(0) := '1';
            ELSIF (reg(0)='1') THEN
                  count := count + 1;
                  IF (count < 10) THEN
                        reg(count) := din;
                  ELSIF (count = 10) THEN
                        temp := (reg(1) XOR reg(2) XOR reg(3) XOR
                        reg(4) XOR reg(5) XOR reg(6) XOR
                        reg(7) XOR reg(8)) OR NOT reg(9);
                        err <= temp;
                        count := 0;
                        reg(0) := din;
                        IF (temp = '0') THEN
                              data_valid <= '1';
                              data <= reg(7 DOWNTO 1);
                        END IF;
                  END IF;
            END IF;
      END IF;
END PROCESS;
END arc;
-------------------------------------------------
```

Kết quả mô phỏng:



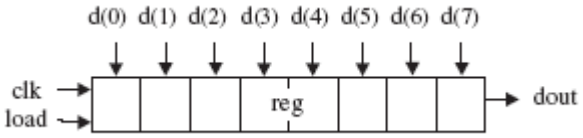
Hình 9.14.Kết quả mô phỏng bộ nhận dữ liệu

9.7. Bộ chuyển song song thành nối tiếp.

Bộ chuyển song song thành nối tiếp là một loại ứng dụng của thanh ghi dịch. Bộ chuyển đổi này sẽ thực hiện việc gửi đi một khối dữ liệu nối tiếp. Việc sử dụng bộ chuyển đổi này là rất cần thiết ví dụ: Trong các con chip ASIC, khi

không có đủ các chân dư để cho ra đồng thời tất cả các bit dữ liệu. Khi đó chúng ta cần thiết phải sử dụng bộ chuyển đổi song song thành nối tiếp.

Sơ đồ khối của bộ chuyển đổi song song thánh nối tiếp được trình bày trong hình 9.15.



Hình 9.15.Bộ chuyển song song thành nối tiếp

Trong đó:

+ d(7:0) là vector dữ liệu để gửi đi
+ dout là đầu ra thực tế.
+ clk: Đầu vào của xung clock
+ load: Đầu vào xác nhận

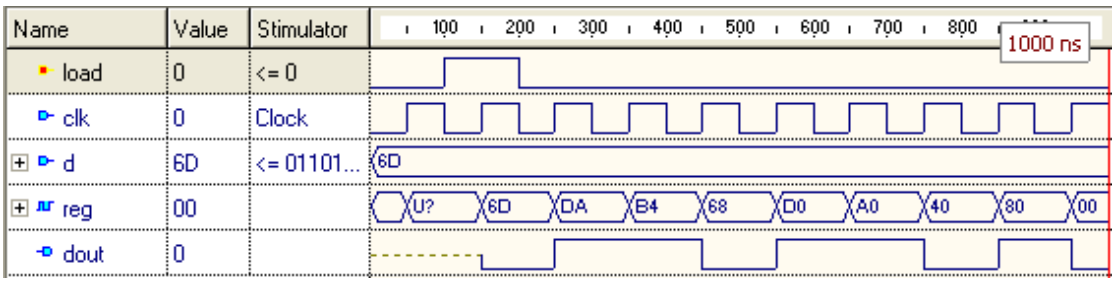
Vector d được lưu trữ đồng bộ trong thanh ghi dịch reg. Khi load ở trạng thái cao thì dữ liệu được nạp vào thanh ghi dịch theo thư tự bit MSB là bít gần đầu ra nhất, và đầu ra là d(7). Mỗi khi load trả lại "0" thì bit tiếp theo được xuất hiện tại đầu ra của mỗi sườn dương của xung đồng hồ. Sau khi tất cả 8 bit được gửi đi, đầu ra trở lại mức thấp cho đến lần chuyển đổi tiếp theo.

Mã thiết kế như sau:

```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-----------------------------------------------
ENTITY Bo_chuyen_dl_ss_nt IS
PORT ( d: IN STD_LOGIC_VECTOR (7 DOWNTO 0);
           clk, load: IN STD_LOGIC;
           dout: OUT STD_LOGIC);
END Bo_chuyen_dl_ss_nt;
-----------------------------------------------
ARCHITECTURE Bo_chuyen_dl_ss_nt OF Bo_chuyen_dl_ss_nt IS
SIGNAL reg: STD_LOGIC_VECTOR (7 DOWNTO 0);
BEGIN
PROCESS (clk)
BEGIN
     IF (clk'EVENT AND clk='1') THEN
          IF (load='1') THEN reg <= d;
          ELSE reg <= reg(6 DOWNTO 0) & '0';
          END IF;
     END IF;
END PROCESS;
dout <= reg(7);
END Bo_chuyen_dl_ss_nt;
```

_____

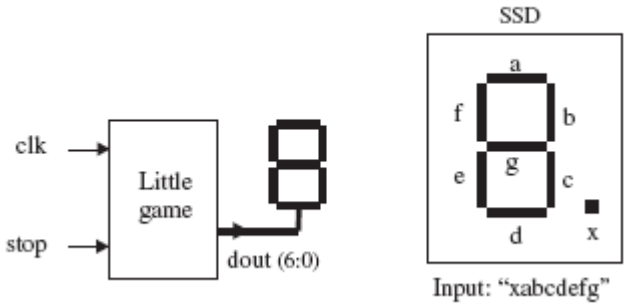
Kết quả mô phỏng:



Hình 9.16. Kết quả mô phỏng cho bộ chuyển song song thành nối tiếp

9.8. Trò chơi trên led 7 thanh.

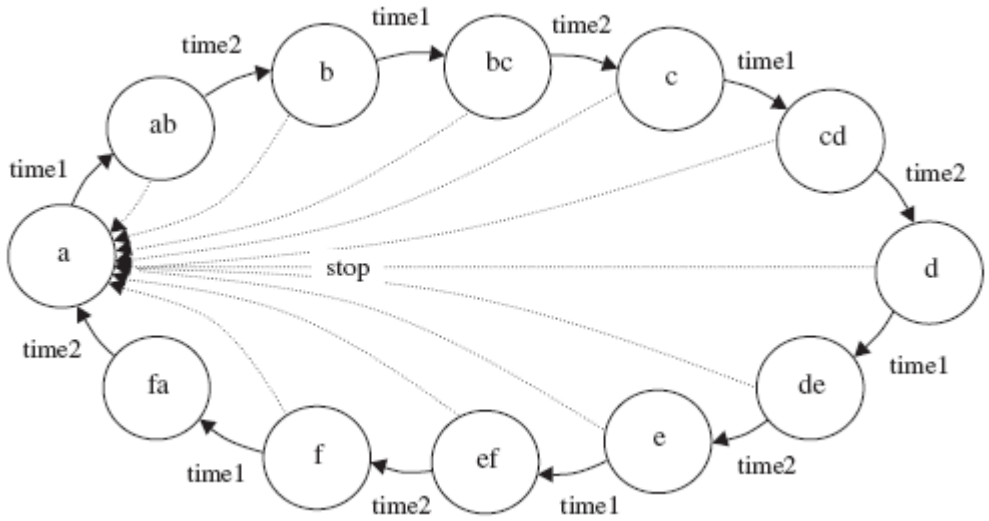
Chúng ta thiết kế trò chơi với SSD (seven – segment display). Sơ đồ của mạch được chỉ ra trong hình 9.17. Nó bao gồm 2 đầu vào là clk và stop, và một đầu ra là dout(6:0), đầu ra này sẽ được hiển thị trên SSD. Chúng ta phải đảm bảo rằng f_{dk} = 1khz



Hình 9.17. Sơ đồ của SSD

Mạch của chúng ta sẽ tạo ra một sự chuyển động liên tục theo chiều kim đồng hồ của các đoạn SSD. Đồng thời nó còn tạo ra sự dịch chuyển chồng lắp giữa các thanh kề nhau. Chúng ta có thể biểu diễn quy trình của nó như sau:

a->ab->b->bc->c->cd->d->de->e->ef->f->fa->a.



Hình 9.18. Đồ hình trạng thái

Quá trình sẽ dừng lại khi có tín hiệu Stop, và khi đó mạch sẽ trở lại trạng thái a và chờ cho đến khi stop xuống thấp trở lại. Hệ thống của chúng ta sẽ giữ lại ở các trạng thái a, b, c, d , e, f trong khoảng thời gian time1 = 80ms và ở các trạng thái ab, bc, cd, de, ef, fa là time2 = 30ms.

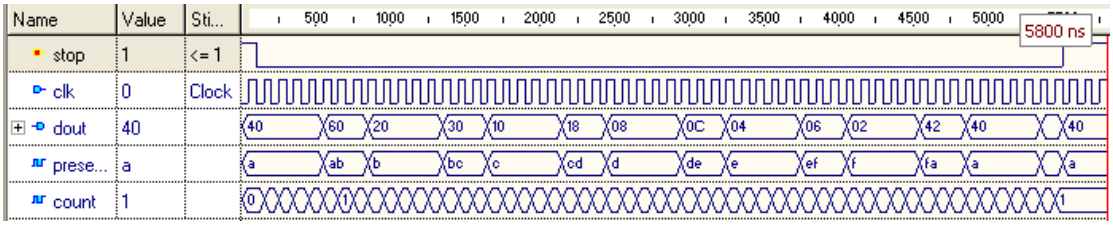
Mã chương trình của chúng ta sẽ như sau:

```
-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY Trochoiled7thanh IS
PORT ( clk, stop: IN BIT;
dout: OUT BIT_VECTOR (6 DOWNTO 0));
END Trochoiled7thanh;
-------------------------------------------------------
ARCHITECTURE arc OF Trochoiled7thanh IS
CONSTANT time1: INTEGER := 4; -- Gia tri thuc te hien thi la
80
CONSTANT time2: INTEGER := 2; -- Gia tri thuc te hien thi is
30
TYPE states IS (a, ab, b, bc, c, cd, d, de, e, ef, f, fa);
SIGNAL present_state, next_state: STATES;
SIGNAL count: INTEGER RANGE 0 TO 5;
SIGNAL flip: BIT;
BEGIN
------- Phan mach day cua arc : ------------
PROCESS (clk, stop)
BEGIN
      IF (stop='1') THEN
            present_state <= a;
      ELSIF (clk'EVENT AND clk='1') THEN
            IF ((flip='1' AND count=time1) OR
                  (flip='0' AND count=time2)) THEN
                  count <= 0;
                  present_state <= next_state;
```

```
            ELSE count <= count + 1;
            END IF;
      END IF;
      END PROCESS;
------- Phan mach to hop: ------------
PROCESS (present_state)
BEGIN
      CASE present_state IS
            WHEN a =>
                  dout <= "1000000"; -- Decimal 64
                  flip<='1';
                  next_state <= ab;
            WHEN ab =>
                  dout <= "1100000"; -- Decimal 96
                  flip<='0';
                  next_state <= b;
            WHEN b =>
                  dout <= "0100000"; -- Decimal 32
                  flip<='1';
                  next_state <= bc;
            WHEN bc =>
                  dout <= "0110000"; -- Decimal 48
                  flip<='0';
                  next_state <= c;
            WHEN c =>
                  dout <= "0010000"; -- Decimal 16
                  flip<='1';
                  next_state <= cd;
            WHEN cd =>
                  dout <= "0011000"; -- Decimal 24
                  flip<='0';
                  next_state <= d;
            WHEN d =>
                  dout <= "0001000"; -- Decimal 8
                  flip<='1';
                  next_state <= de;
            WHEN de =>
                  dout <= "0001100"; -- Decimal 12
                  flip<='0';
                  next_state <= e;
            WHEN e =>
                  dout <= "0000100"; -- Decimal 4
                  flip<='1';
                  next_state <= ef;
            WHEN ef =>
                  dout <= "0000110"; -- Decimal 6
                  flip<='0';
                  next_state <= f;
            WHEN f =>
                  dout <= "0000010"; -- Decimal 2
                  flip<='1';
                  next_state <= fa;
            WHEN fa =>
                  dout <= "1000010"; -- Decimal 66
                  flip<='0';
```

```
                    next_state <= a;
      END CASE;
END PROCESS;
END arc;
-------------------------------------------------------
```

Kết quả mô phỏng:

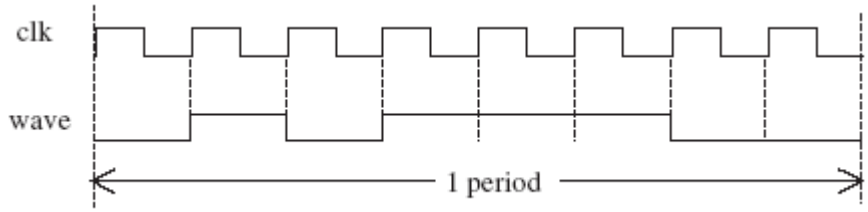


Hình 9.19. Kết quả mô phỏng cho trò chơi trên SSD

9.9. Bộ phát tín hiệu.

Từ một tín hiệu clock, chúng ta mong muốn thu được một tín hiệu có dang sóng như trong hình 9.20. Với bài toán loại này, chúng ta có thể sử dụng phương pháp FSM hoặc phương pháp truyền thống. Cả 2 phương pháp đều được chúng ta trình bày dưới đây:

Phương pháp FSM:



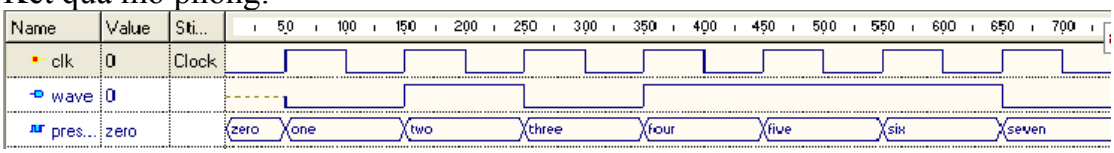
Hình 9.20 Hình dạng sóng cần phát

Tín hiệu của hình 9.20 có thể được mô hình như một FSM 8 trạng thái. Sử dụng bộ đếm từ 0 đến 7. Chúng ta có thể thiết lập một sóng bằng ‘0’ khi biến đếm = ‘0’ (ở xung thứ nhất) và bằng 1 khi biến đếm = ‘1’ (xung thứ hai),…vv…như trong hình 9.20. Để thực thi được bộ tạo sóng này thì yêu cầu 4 flip-flop: trong đó có 3 cái để lưu trữ số đếm (3 bit), một cái để lưu trữ sóng (1 bit). Để thiết kế bộ tạo sóng này, chúng ta thiết kế theo kiểu 2, cụ thể sẽ như sau:

```
-------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------------
ENTITY Bo_phat_tin_hieu IS
PORT (clk: IN STD_LOGIC;
      wave: OUT STD_LOGIC);
END Bo_phat_tin_hieu;
```

```
-------------------------------------------------
ARCHITECTURE arc OF Bo_phat_tin_hieu IS
TYPE states IS (zero, one, two, three, four, five, six,
seven);
SIGNAL present_state, next_state: STATES;
SIGNAL temp: STD_LOGIC;
BEGIN
--- Phan mach day: ---
PROCESS (clk)
BEGIN
      IF (clk'EVENT AND clk='1') THEN
            present_state <= next_state;
            wave <= temp;
      END IF;
END PROCESS;
--- Phan mach to hop: ---
PROCESS (present_state)
BEGIN
      CASE present_state IS
            WHEN zero => temp<='0'; next_state <= one;
            WHEN one => temp<='1'; next_state <= two;
            WHEN two => temp<='0'; next_state <= three;
            WHEN three => temp<='1'; next_state <= four;
            WHEN four => temp<='1'; next_state <= five;
            WHEN five => temp<='1'; next_state <= six;
            WHEN six => temp<='0'; next_state <= seven;
            WHEN seven => temp<='0'; next_state <= zero;
      END CASE;
END PROCESS;
END arc;
-------------------------------------------------
```

Kết quả mô phỏng:



Hình 9.2.1. Kết quả mô phỏng tạo sóng

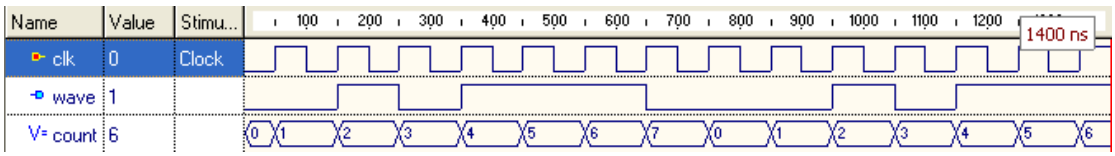
Phương pháp truyền thống:

Chúng ta thiết kế bộ phát tín hiệu theo phương pháp truyền thống với câu lệnh IF như sau:

```
---------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
---------------------------------------
ENTITY Bo_phat_tin_hieu2 IS
PORT (clk: IN BIT;
      wave: OUT BIT);
END Bo_phat_tin_hieu2;
---------------------------------------
ARCHITECTURE arc OF Bo_phat_tin_hieu2 IS
BEGIN
PROCESS
```

```
VARIABLE count: INTEGER RANGE 0 TO 7;
BEGIN
      WAIT UNTIL (clk'EVENT AND clk='1');
      CASE count IS
            WHEN 0 => wave <= '0';
            WHEN 1 => wave <= '1';
            WHEN 2 => wave <= '0';
            WHEN 3 => wave <= '1';
            WHEN 4 => wave <= '1';
            WHEN 5 => wave <= '1';
            WHEN 6 => wave <= '0';
            WHEN 7 => wave <= '0';
      END CASE;
      if count = 7 then
            count := 0;
      else
            count := count + 1;
      end if       ;
END PROCESS;
END arc;
---------------------------------------
```

Kết quả mô phỏng:



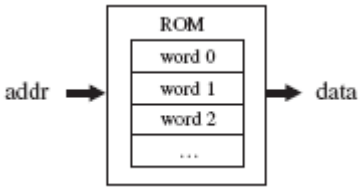
Hình 9.22. Kết quả mô phỏng tạo sóng theo phương pháp truyền thống

9.10. Thiết kế bộ nhớ.

Trong đoạn này, chúng ta sẽ thiết kế các mạch bộ nhớ sau:

+ ROM
+ RAM với bus dữ liệu vào ra tách rời.
+ ROM với bus dữ liệu vào ra hai chiều

ROM (Read Only Memory): Bộ nhớ chỉ đọc và ghi: Sơ đồ của ROM được chỉ ra trong hình 9.23. Vì ROM là bộ nhớ chỉ đọc, không có tín hiệu clock, chân cho phép ghi, nó chỉ có tín hiệu vào bus địa chỉ và tín hiệu ra là bus dữ liệu.

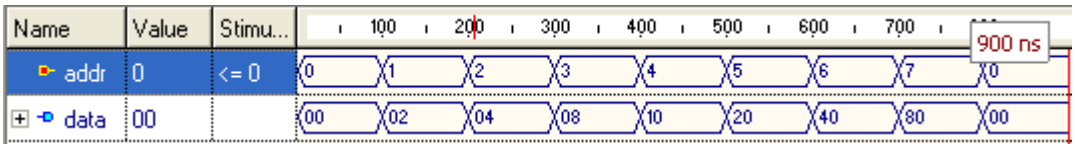


Hình 9.23.Sơ đồ của ROM

Mã thiết kế ROM như sau:

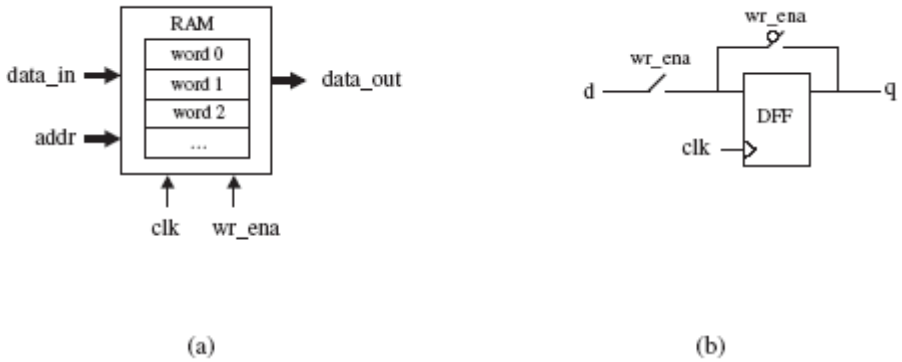
```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY rom IS
GENERIC ( bits: INTEGER := 8; -- # of bits per word
          words: INTEGER := 8); -- # of words in the memory
PORT ( addr: IN INTEGER RANGE 0 TO words-1;
          data: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END rom;
-------------------------------------------------
ARCHITECTURE rom OF rom IS
TYPE vector_array IS ARRAY (0 TO words-1) OF
STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
CONSTANT memory: vector_array := ( "00000000",
                                   "00000010",
                                   "00000100",
                                   "00001000",
                                   "00010000",
                                   "00100000",
                                   "01000000",
                                   "10000000");
BEGIN
     data <= memory(addr);
END rom;
-------------------------------------------------
```

Kết quả mô phỏng:



Hình 9.24. Kết quả mô phỏng thiết kế ROM

RAM với đường bus vào ra riêng biệt: Sơ đồ của RAM với đường bus vào ra riêng biệt được thể hiện trong hình 9.25



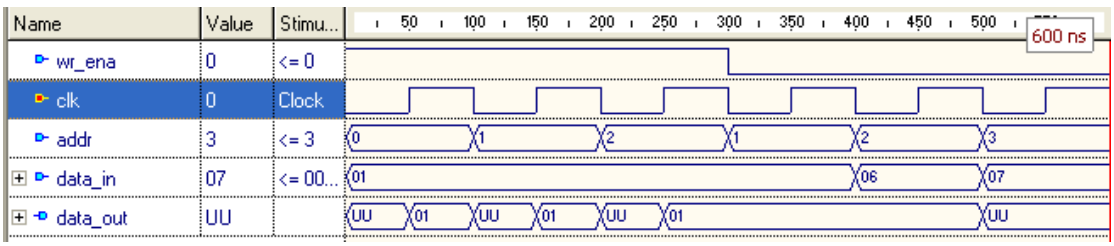
Hình 9.25. RAM với đường dữ liệu tách rời

Như chúng ta thấy trên hình, RAM có các bus dư liệu vào data_in, bus dữ liệu ra data_out, bus địa chỉ, tín hiệu clk và tín hiệu cho phép đọc/ghi. Khi tín hiệu cho phép ghi/đọc được xác nhận là ghi thì tại mỗi xung lên tiếp theo của clk thì dữ liệu đầu vào (data_in) phải được lưu trữ tại vị trí addr, và dữ liệu ra phải được đọc từ địa chỉ addr.

Mã thiết kế RAM sẽ như sau:

```
------------------------------------------------------------
---------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
-------------------------------------------------
ENTITY ram IS
GENERIC ( bits: INTEGER := 8; -- # of bits per word
      words: INTEGER := 16); -- # of words in the
-------- memory----------
PORT ( wr_ena, clk: IN STD_LOGIC;
           addr: IN INTEGER RANGE 0 TO words-1;
           data_in: IN STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
           data_out: OUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END ram;
-------------------------------------------------
ARCHITECTURE ram OF ram IS
TYPE vector_array IS ARRAY (0 TO words-1) OF
STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
SIGNAL memory: vector_array;
BEGIN
PROCESS (clk, wr_ena)
BEGIN
      IF (wr_ena='1') THEN
            IF (clk'EVENT AND clk='1') THEN
                  memory(addr) <= data_in;
            END IF;
      END IF;
END PROCESS;
data_out <= memory(addr);
END ram;
-------------------------------------------------
```

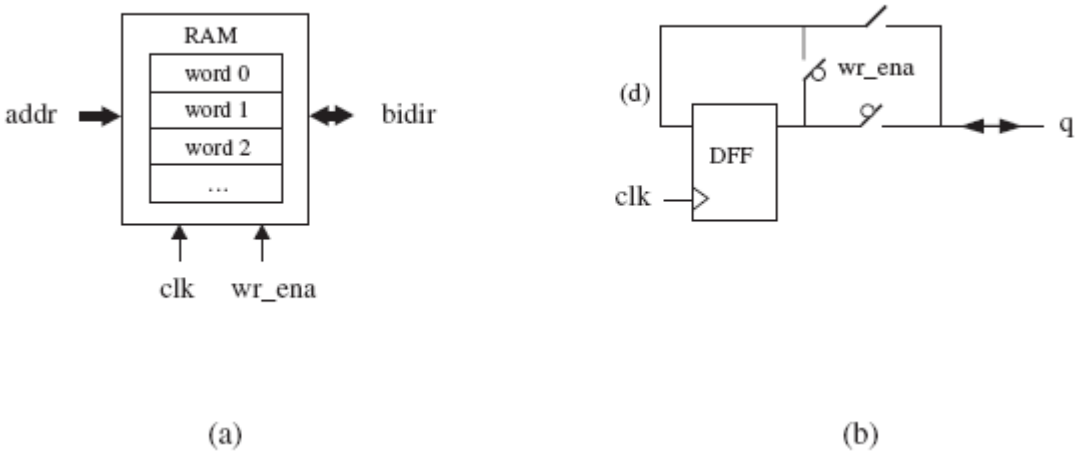
Kết quả mô phỏng:



Hình 9.26. Kết quả mô phỏng RAM có đương dữ liệu vào ra khác nhau.

RAM với đường bus song song:

Sơ đồ của RAM với đường bus song song được thể hiện trong hình 9.27. Dữ liệu được ghi vào RAM hay được đọc từ RAM thực hiện trên cùng 1 đường bus.



Hình 9.27. RAM với đường dữ liệu chung

Mã thiết kế sẽ như sau:

```
-------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
-------------------------------------------------
ENTITY ramc IS
GENERIC ( bits: INTEGER := 8; -- # of bits per word
      words: INTEGER := 16); -- # of words in the
-- memory
PORT ( clk, wr_ena: IN STD_LOGIC;
           addr: IN INTEGER RANGE 0 TO words-1;
           bidir: INOUT STD_LOGIC_VECTOR (bits-1 DOWNTO 0));
END ramc;
-------------------------------------------------
ARCHITECTURE arc OF ramc IS
TYPE vector_array IS ARRAY (0 TO words-1) OF
STD_LOGIC_VECTOR (bits-1 DOWNTO 0);
SIGNAL memory: vector_array;
BEGIN
PROCESS (clk, wr_ena)
BEGIN
      IF (wr_ena='0') THEN
            bidir <= memory(addr);
      ELSE
            bidir <= (OTHERS => 'Z');
            IF (clk'EVENT AND clk='1') THEN
                  memory(addr) <= bidir;
            END IF;
      END IF;
END PROCESS;
END arc;
-------------------------------------------------
```

Kết luận

Ngày này việc ứng dụng VHDL trong việc thiết kế mạch và chíp ngày cang nhiều. Công nghệ này đang là xu hướng của thời đại, đơn giản vì nó không chỉ tiêu tốn ít về tiền bạc mà nó còn giúp cho chúng ta đơn giản trong việc thiết kế phần cứng.

Trên đây, chúng ta đã trình bày một cách khái quát về phương pháp thiết kế các mạch. Những mạch cơ bản nhất đã được chúng ta thiết kế một cách chi tiết, hoàn thiện. Đây là cơ sở cho những thiết kế lơn hơn về phần cứng, để thiết kế các ứng dụng cho các FPGA, ASIC.

Tài liệu tham khảo:

- Circuit design with VHDL , Voilnei A.Pedroni
- VHDL language.
- The vhdl – cookbook , Peter J.Ashedo
- Thiết kế mạch bằng máy tính, Nguyễn Linh Giang

…..

Phân công công việc:

Nguyễn Ngọc Linh: Chương 2,3
Nguyễn Quốc Việt: Chương 4,5

Nghiêm Kim Phương: Chương 6,7
Lê Tuấn Anh: Chương 8, 9, giới thiệu, tổng kết